Higher Order Software, Inc.

Technical Report TR-4

LEVEL II

AXES Syntax Description

by

M. Hamilton and S. Zeldin

with Appendices III and IV: S. Cushing

and Appendix V: W. Heath

Prime Contractor:

THE CHARLES STARK DRAPER
LABORATORY, INC.
555 Technology Square
Cambridge, MA 02139

Subcontractor:

HIGHER ORDER SOFTWARE, INC.
843 Massachusetts Avenue
Cambridge, MA 02139

December 1976

Prepared for: Naval Electronics Laboratory Center
San Diego, CA 92152

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: TR-4
2. GOVT ACCESSION NO.: AD-A095988
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): AXES Syntax Description
5. TYPE OF REPORT & PERIOD COVERED: Final Report for Period Sep 75 – Dec 76
6. PERFORMING ORG. REPORT NUMBER: Higher Order Software Technical Report #4

7. AUTHOR(s): M. Hamilton, S. Zeldin, J. Cushing, W. Heath
8. CONTRACT OR GRANT NUMBER(s): N00123-75-C-1636

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Prime Contractor:
The Charles Stark Draper Laboratory
555 Technology Square
Cambridge, MA 02139

Subcontractor:
Higher Order Software, Inc.
643 Massachusetts Avenue
Cambridge, MA 02139

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 133

11. CONTROLLING OFFICE NAME AND ADDRESS

12. REPORT DATE: December 1976
13. NUMBER OF PAGES: 140

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
Naval Electronics Laboratory Center
San Diego, CA 92152

15. SECURITY CLASS. (of this report): UNCLASSIFIED
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

formal systems, reliability, axioms, specification, control, methodology, decomposition, syntax, semantics, algebraic specification, structures, operations, functions, HOS

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This document is intended to serve as an introduction to the specification language AXES. The foundations of AXES (HAM76a) are based on the formal system theory of Higher Order Software. This document contains the definition of the syntax of AXES. In addition, semantics definitions in AXES (e.g., primitive control structures, intrinsic data type definitions) are described. Examples of system specifications are used to illustrate various features of the language.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ACKNOWLEDGEMENT

The AXES project was directed by Margaret Hamilton and Saydean Zeldin. Major technical contributions were provided by Steven Cushing, especially in the area of the semantic preliminaries (Section 2.0) and the specification of the abstract data types. Appendix V, A Sample AXES System Definition, was contributed by Bill Heath.

Higher Order Software, Inc. would like to thank Marty Wolfe of CENTACS, ECOM/Fort Monmouth, New Jersey for his very helpful ideas related to AXES as a result of our many design sessions during the course of our ISDS/HOS project. We would also like to thank Robert Freiburghouse from Translation Systems, Inc. for his suggestions made during the earlier part of this project.

Our project monitor, Tricia Santoni, has provided us with many constructive comments throughout the duration of this project.

We would also like to express appreciation to Andrea Davis and Gail Lopes for the preparation of this report.

We performed our preliminary work at The Charles Stark Draper Laboratory prior to the incorporation of Higher Order Software, Inc.

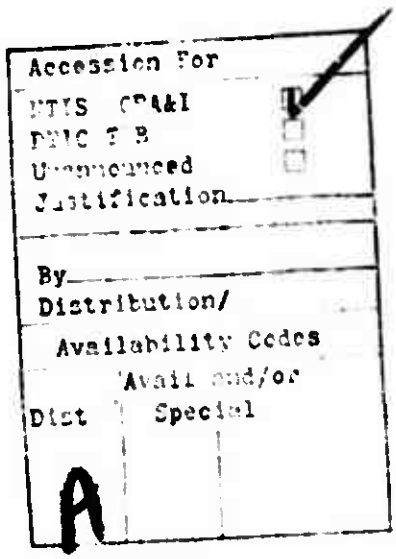


TABLE OF CONTENTS

		Page
1.0	INTRODUCTION	1
2.0	SEMANTIC PRELIMINARIES AND A NOTATIONAL CONVENTION	2
3.0	OBJECTS OF SPECIFICATION	8
4.0	SPECIFICATION VS. IMPLEMENTATIONAL LANGUAGE	12
5.0	NOMENCLATURE	15
6.0	COMMENTS	16
7.0	MULTI-LINE FORMAT	16
8.0	ABSTRACT CONTROL STRUCTURES	18
9.0	DECLARATIONS	23
10.0	DEFINITIONS	27
11.0	EXPRESSIONS	29
12.0	PRIMITIVE ABSTRACT CONTROL STRUCTURES AS STRUCTURES	32
13.0	THE USE OF STRUCTURES, OPERATIONS AND FUNCTIONS	34
14.0	FUNCTION FAILURE	38
15.0	DATA TYPES	40
16.0	DERIVED OPERATIONS	45
17.0	RECURSION VS. ITERATION	47
18.0	CONCLUDING REMARKS	49
BIBLIOGRAPHY		51
APPENDIX I:	PRELIMINARIES OF HOS	
APPENDIX II:	PROPERTIES OF THE PRIMITIVE CONTROL STRUCTURES	
APPENDIX III:	ALGEBRAIC SPECIFICATION OF ABSTRACT DATA TYPES	
APPENDIX IV:	THE INTRINSIC TYPES OF AXES	
APPENDIX V:	SAMPLE AXES SYSTEM SPECIFICATION	
APPENDIX VI:	SUMMARY OF THE DESCRIPTION OF AXES SYNTAX	

HIGHER ORDER SOFTWARE, INC. · 843 MASSACHUSETTS AVENUE · CAMBRIDGE, MASSACHUSETTS 02139 · (617) 661-8900

1.0 INTRODUCTION

The Specification Language, AXES, is a formal notation for writing functional definitions of systems. Although it is not a programming language, AXES is a complete and well-defined language capable of being analyzed by a computer.

Higher Order Software (HOS) (HAM76b) is a formal system theory that forms the foundations of the Specification Language, AXES. HOS is a unified systems-engineering methodology that encompasses all phases and all disciplines of computer-based systems development. With the methodology of HOS, we apply the same axioms (Appendix I) and therefore the same decomposition techniques throughout an integrated system development (HAM76c). AXES is the tool for defining and describing functions and interfaces of a system throughout all phases of a system development.

The purpose of AXES is to be able to express a specification in a form which is equivalent to an HOS control map (HAM76a). Thus, systems described in AXES are based on the use of three primitive control structures, which were derived from the HOS axioms (Appendix II). With the syntax of AXES, we are able to describe a system using the primitive control structures, intrinsic data types and universal primitive operations of AXES, or we have the option of defining new data types or new control structures which can also be used to describe a system.

A computer-based AXES analyzer can be developed in order to check the consistency and completeness of functions described within AXES statements.

HIGHER ORDER SOFTWARE, INC. · 843 MASSACHUSETTS AVENUE · CAMBRIDGE, MASSACHUSETTS 02139 · (617) 661-8900

2.0 SEMANTIC PRELIMINARIES AND A NOTATIONAL CONVENTION

Throughout our description of AXES, we will be using a notation that is conventional in semantics, but that may not be familiar to most readers. One of the most fundamental insights of semantics is the fact that we can talk about an object only by using a name of the object. To talk about the man in Figure 2.1, for example, we have to use a sentence that contains the man's name, not the man himself.

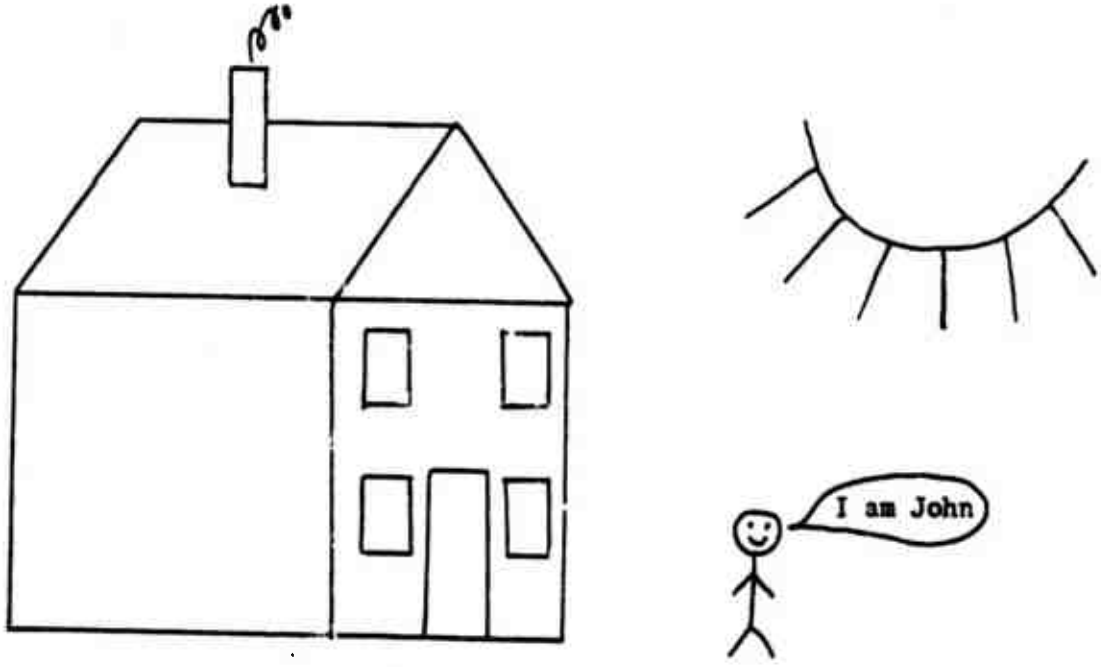


Figure 2.1

The sentence

(1) John is standing beside the house

is a true description of the situation pictured in Figure 2.1, but the purported sentence

(2)  is standing beside the house

is not a sentence at all, because the man himself appears in it, rather than his name. How we say things about objects is always one step removed from the objects themselves.

This fact is so basic a part of how we communicate information that we generally remain entirely unaware of it, taking it entirely for granted like a pulse or heartbeat. When it is brought to one's attention, it may even seem trival or unimportant. Serious problems can arise, however, when we begin talking about a language, such as AXES, rather than simply using a language to talk about objects. Since how we say something must of necessity be one step removed, linguistically, from what we are saying about it, great care must be taken to distinguish the names in the language we are talking about from the names in the language we are using.

When we are talking about a language, we are treating the names of that language as objects. We can only talk about those objects (names), as with any objects, by using names of them. It follows that we need a notation for names of names, if we intend to talk consistently about names. The notation conventionally used for this in semantics is enclosure within quotation marks. To form the name of a given name (or written symbol of any kind) we include that name (or symbol) in quotation marks.

We can clarify this notation somewhat by examining a few examples. Consider the man in Figure 2.2 and the four purported sentences in (3) and (4):

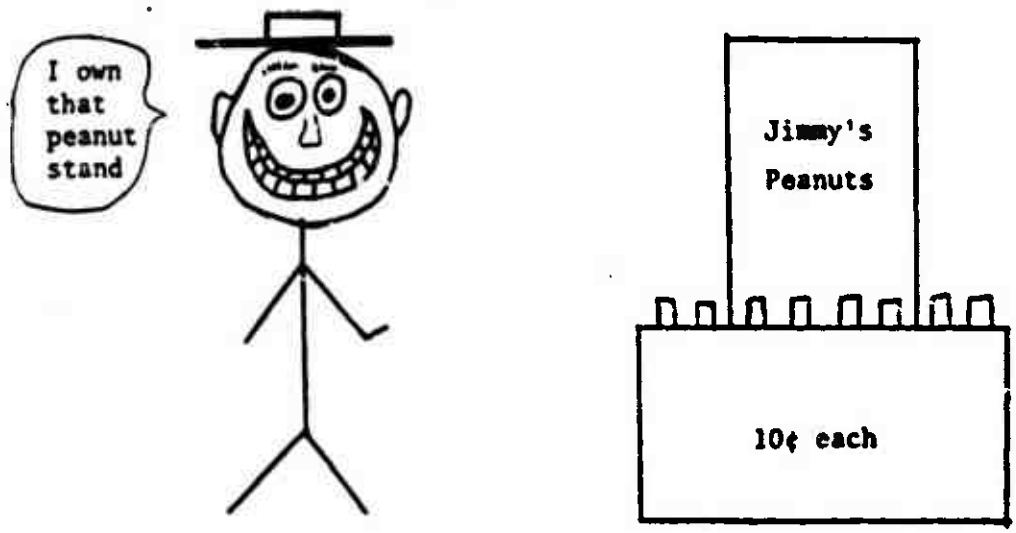


Figure 2.2

(3) (a) Jimmy sells peanuts
(b) * Jimmy is bisyllabic

(4) (a) * "Jimmy" sells peanuts
(b) "Jimmy" is bisyllabic

In both pairs of purported sentences (3) and (4), those which are prefixed with asterisks (*) are not really sentences at all, but meaningless strings of words, while those without asterisks are normal meaningful sentences which also happen to be true. Sentence (3a) uses the man's name to talk about the man, saying that the man sells peanuts. Purported sentence (4a), in contrast, is not using a name of the man, but a name of the man's name, since the name it uses as its subject is the man's name in quotes. Since (4a) uses the name of a name, it is talking about a name, saying that that name sells peanuts, an obvious absurdity. Sentence (4b), however, is all right, because, while also talking about a name, what it says about that name makes sense. Names cannot sell peanuts, but they can be bisyllabic. Purported sentence (3b), conversely, is an absurdity, like (4a). By using the name of a man it talks about the man himself, saying that he is bisyllabic, which makes no sense.

Successive embedding of quotation marks can be used if we want to talk about names, names of names, names of names of names, etc., as illustrated in Figure 2.3.

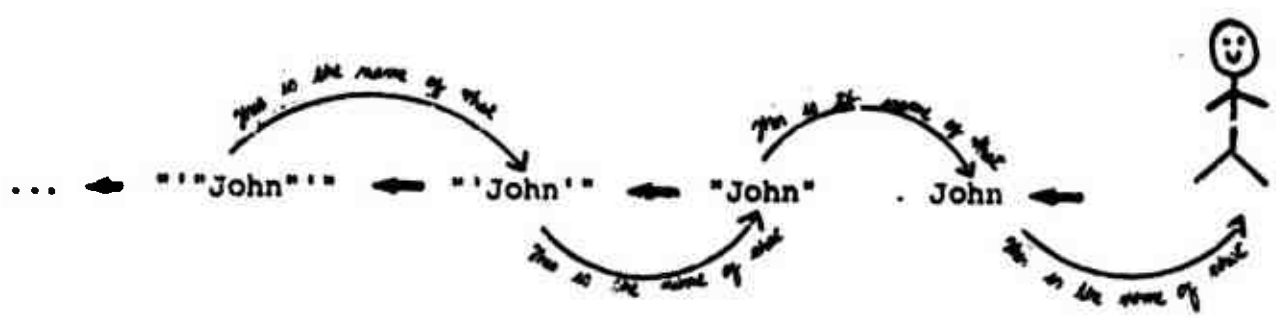


Figure 2.3

If we begin at the right and move leftward, we first have our object, the man, and we then get his name. To talk about the man, we must use his name. We are then free to view that name as an object and we get its name by moving one more step to the left. To talk about the name we must use its name (i.e., the thing in quotation marks). If necessary we can then treat that name as an object and talk about it using its name, obtained by moving still one more step to the left. What we obtain then is the name of the name of the name of the man, which we use to talk about the name of the name of the man. The process can be continued indefinitely, in principle, but it is unlikely that we would ever have to go beyond the steps shown in the diagram, in actual practice.

The main point to be kept in mind is the need to distinguish carefully between an object and its name and to make sure that we use the name, not the object itself, to talk about the object. In (3) and (4) we saw how confusing the object with its name can turn a seemingly normal sentence into an absurdity. Sometimes, however, it can produce a perfectly meaningful sentence whose actual meaning differs from what it was intended to mean. Each of the sentences

(5) Jimmy sounds funny
(6) "Jimmy" sounds funny

is a perfectly meaningful sentence, but their meanings are very different. Sentence (5) makes sense if completed with

(7) in contrast with Northerners

but sentence (6) must be completed with something like

(8) because it's really just a nick-name

to make sense.

If we add (7) to (6) or (8) to (5), we get meaningless nonsense like (3b) and (4a).

The reason this principle is important for us, of course, is that we are describing a language, AXES, and we must be careful that what we say about that language makes sense. We will be talking about the names and other symbols of AXES, i.e., we will be treating them as objects, so we must be careful to use their names in doing so. The quotation-mark convention enables us to form the names of the AXES names and thus to talk about the AXES names themselves in a consistent way.

In AXES what corresponds to names are variables and constant symbols. A constant symbol is the name of a particular value and corresponds to a proper name like "John." A variable is the name of more than one possible value and corresponds to a common noun like "a man." Figure 2.4a exhibits a number of constant symbols and Figure 2.4b exhibits some variables.

(a) (b)

Figure 2.4

Note that the quotation mark convention applies as much to common nouns as to proper names, so the sentences

A man has two arms
"A man" contains two words

make sense, while the purported sentences

"A man" has two arms
A man contains two words

do not.

In describing AXES we will use variables and constants themselves to make statements about the values they name, and we will use the names (quoted forms) of variables and constants to make statements about the variables and constants themselves. The sentences

x = y+z
w = 3
x is an integer
y and z are of the same type,

for example, make statements about values by using the variables and constant symbols that name those values. Sentences like

"x" represents the same value as "y+z"
"w" represents a value of type integer
"y" and "z" represent values of the same type
"q" is a variable and "3" is a constant symbol,

in contrast, make statements about variables and constant symbols by using the quotation-marked symbols that name them. In describing AXES we will try to adhere scrupulously to this convention, so that it will always be clear whether we are talking about the objects (values, functions, mappings, structures, etc.) that AXES talks about or whether we are talking about the variables and other symbols that make up AXES itself. (Appendix VI)

3.0 OBJECTS OF SPECIFICATION

An AXES system is a control hierarchy. The structure of any hierarchy is determined by the objects that belong to the hierarchy and the relationship that exists between the objects of the hierarchy. For AXES systems, the objects are variables, values, functions, and trees; the relationship is control.

An AXES system can be graphically represented as a tree in which each node identifies a member of a given control hierarchy. An AXES tree structure, called a control map, is a relation on a set of mappings, i.e, a set of tuples whose members are sets of ordered pairs. An invocation tree, illustrated in Figure 3.1, exhibits the names of the sets of ordered pairs (i.e., mappings) which complete the functional specification for System A. When our intent is to understand or describe the relation on the set of mappings, the corresponding function of System A is described as a decomposition of A into levels. The most immediate lower level of A is a realization of A and only A. Functions A_1 and A_2 are on the first lower level of A, and functions B_1 and B_2 are on the second lower level of A with respect to A_2.

A controls the use of A_1 and A_2; A_2 controls the use of B_1 and B_2. The properties of control are determined by the axioms of HOS. When we refer to A controlling A_1 and A_2, A is referred to as a controller or as a module. When we refer to A_1 with respect to A, A_1 is referred to as a function.

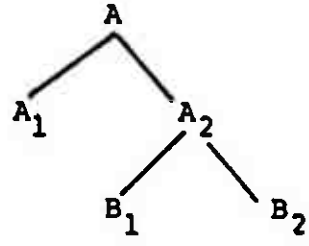


Figure 3.1: System A Invocation Tree

A control map includes a description of each node in terms of the input/output representation of each function as well as the name associated with each node of the hierarchy. A control map is constructed using abstract control structures (see Section 8.0). A control map of System A is shown in Figure 3.2.

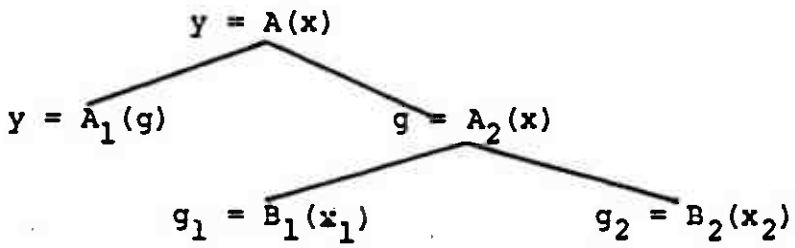


Figure 3.2: A Control Map of System A

A representation of the input value or output value of a function is called an input or output variable, respectively. In (3-1), "x" is an input variable of A; "y" is an output variable of A.

$$y = A(x) \qquad (3\text{-}1)$$

Suppose "x" represents any one of the integers 5, 8, or 2. We refer to these integers as <u>values</u> of "x". Likewise, if "y" represents any one of the integers 6, 10, or 2, we refer to these integers as values of "y".

In an AXES system, a <u>function</u> refers to the relationship (i.e., a mapping) between the input values and the output values where these values are represented by particular variables (c.f. FUNCTION definition Section 8.0). This relationship is restricted in AXES so that any input value corresponds to one and only one output value.

A <u>data type</u> is a set of values characterized by a set of primitive operations. When a variable is of a data type, that variable represents a value of that data type. A variable of a data

type may be replaced by a set of variables whose values collectively represent the same value as the variables of the data type. This collective set of variables is the data structure of the variable of the data type.

Suppose "y" in (1) is replaced by "y_1" and "y_2", and "x" by "x_1" and "x_2". The data structure of x is (x_1,x_2), and the data structure of y is (y_1,y_2). The data structures of x and y could be used by a function, such as B, to accomplish the same mapping as System A.

$$(y_1,y_2) = B(x_1,x_2) \qquad (3\text{-}2)$$

If x is of data type 2-tuple integer, then (1,10) is a value of "x". If y is of data type 2-tuple integer, then (1,10) is also a value of "y." "x" and "y" represent the same set of values, i.e., "x" and "y" are variables that represent values of the same data type. A and B are equivalent functions.

When our intent is to use a system as the input variable or output variable of another system, A is of data type system and the names of the functions on the first immediate lower level of A (i.e., A_1 and A_2) describe the data structure of System A. Similarly, B_1 and B_2 are the data structure of A_2. When "A" is considered an input or output variable of another system, "A" represents a layer.

There are many tradeoffs that must be considered in developing the layers of a system. These involve not only how many layers, but whether or not these layers are created statically (development layers) or dynamically (execution layers).

If, for example, a translator (such as a compiler) converted one description of a specification of A to another description of a specification of A, System A would exist as at least two development layers. If, however, a real-time translator (such as an

operating system) converted a System A layer to an executable mode in real time, we would call that new layer of A an execution layer.

In defining a system it may be necessary to describe the system both as a function and as a layer. The syntax of AXES provides the means to differentiate between these two concepts.

The intent of an AXES specification is to describe the functional, data, and performance aspects of a system independently of a particular resource allocation (i.e., implementation). The functional description provides the specification of the decomposition of a system; the data description provides the specification for the data types and structures to be used in the functional specification; the performance description asserts the limitations or constraints associated with the use of functional or data descriptions. With AXES we are able to define systems in which resource allocation requirements (e.g., time and memory) can be specified, thus allowing for resource allocation alternatives. Each of these aspects of a system can be documented in a standard way with AXES.

4.0 SPECIFICATION VS. IMPLEMENTATION LANGUAGE

Using AXES, a system designer describes a system as a set of functions and data that look very much like the procedures and data of a programming language. However, the functions and data of AXES differ in fundamental ways from the procedures and data of programming languages.

In programming languages, a variable "x" is a name that designates a unit of storage where values may be placed. That is a statement of the form "x = y+z" means, "add the current values of 'y' and 'z' and store the result in 'x'". In AXES, the same statement means "'x' represents the same value as is represented by 'y+z'". In AXES, a variable is the name of a particular unspecified value. A constant symbol, such as "2", for example, is, in contrast, the name of a particular specified value. The meaning of "x=y+z;" in AXES differs from its meaning in programming languages also as a result of a difference in meaning of "=". In a programming language, "=" is a directional symbol meaning "is to be replaced with." The statement "x+y=z" means, in effect, "replace whatever value is stored in 'x' with the result of adding whatever value is stored in 'y' to whatever value is stored in 'z'." In AXES, however, "=" is a non-directional symbol meaning "is the same as." The statement "x=y+z" means "the value of 'x' is the same as the sum of the value of 'y' and the value of 'z'," or equivalently, "'x' represents the same value as 'y+z' represents." The interpretation of "=" in AXES is thus identical to the interpretation of "=" in mathematics. AXES statements are statements of fact; they are not commands to be performed.

Programming languages make use of the notion of an order of evaluation or a flow of execution. There is no such notion in AXES. In AXES we use variables to specify equality relationships among values. AXES serves to define the variables that represent values in terms of other defined variables that represent values. In AXES, the following statements simply define the variables "x", "w", and "z" by using them in statements of equality.

```
x = a+b;                                    (4-1)
w = 5;                                      (4-2)
z = f(k);                                   (4-3)
```

Each of these variables represent one of a set of values. If statements (4-1),(4-2),(4-3) were part of an AXES system specification, (4-1) could appear after (4-3) or (4-2), because AXES statements can appear in any order.

In AXES, the following statement defines the variable "x" by using the constant symbol "True".

```
x = AND(True,True)                          (4-4)
```

In AXES, the following statement defines a relationship among systems.

```
Sum(Prod(x,y),x) = Opp(x,Diff(x,y)          (4-5)
```

In this statement, "x" always represents the same value, and "y" always represents the same value.

In AXES, a variable is specified to be referenced only once for a given change of state at a given level of a control hierarchy. (This is called single reference.) A variable is specified to be assigned only once for a given change of state at a given level of a control hierarchy. (This is called single assignment). Thus, the concept of sharing *locations*, is not assumed; yet, this concept may still be introduced into an implementation model.

Functions are defined in such a way that the ordering between functions in a given system can be determined. The *procedures* in an implementation model can thus exist within an unlimited multiprocessor system, a multiprogramming system or a sequential programming system.

Each function is explicitly shown as unique in AXES. Yet, the concept of sharing *instructions* may still be introduced into an implementation model.

Each AXES function is specified to be initiated upon receipt of its first input value. An AXES function is ready for complete execution upon receipt of all of its input values and is completed upon receipt of all its output values. In an AXES system, the specification of a value is synonomous with the specification of an event. Thus, interrupts or searches for *events* are not assumed; yet, they may still be introduced into an implementation model.

5.0 NOMENCLATURE

In the description of AXES the following nomenclature will be used.

":=" means "is".

"{ }" means choose one of the rows contained within.

"[ ]" means the enclosed is optional.

"..." means repeat with different values as often as necessary.

In the syntax of AXES, the following nomenclature will be used.

Upper case names will designate lexical items of AXES (keywords).

"set of variables" means a list of variables possibly enclosed in parentheses.

Constants and abstract control structure names begin with an upper case character followed by zero or more lower case characters.

A variable is indicated by all lower case characters.

A value of a particular data type can be indicated by the name of the data type in lower case characters, possibly subscripted.

6.0 COMMENTS

Comments can be inserted between statements. A comment is delimited at the start by the character pair /*, and at the end by the character pair */. Any character may appear in the comment (except for * followed by /).

7.0 MULTI-LINE FORMAT

A variable in AXES can be a subscripted symbol, a superscripted symbol, or an unsubscripted or unsuperscripted symbol.

AXES allows a multi-line format (LIC74) (LAN52) corresponding to natural mathematical notation. For example, the following statements are acceptable in AXES.

$$y_{i_{t_2}} = F(x_{k_{t_1}})$$

$$^{1}y_{p(t)} = F(x,t)$$

$$a = b^{2} + c^{2^{n}} + d_{e_3}$$

Subscripts

The subscripting of a variable in AXES always signifies a mapping between the value of the subscript variable and the value of the variable that is subscripted. Thus "A_i" shows a relationship between i and A. If that relationship is function F, such that

$$A_i = F(i)$$

"i" could represent an index for memory slot "A_i".

If "x_t" relates x and t by function G, i.e.,

$$x_t = G(t)$$

"t" could represent an index for time slot "x_t". The subscript mechanism is helpful in functionally representing the specifications of resource allocation of storage and time with respect to a particular system.

Superscripts

A variable with a left superscript represents a member of a member of a partition of a set of values. For example, if x is an integer, the members of the sets that make up a partition of the set of which x is a member might be represented by "^1x" and "^2x" where "^1x" represents values greater than 10, and "^2x" represents values less than or equal to 10.

A variable with a right superscript is an alternate notation associated with particular operations on intrinsic data types of AXES (for example, "x^2" means "multiply the value of 'x' by the value of 'x'"). In other words, right superscripts represent mathematical exponents.

8.0 ABSTRACT CONTROL STRUCTURES

An abstract control structure (ACS) is a control hierarchy. An ACS can be generalized to be used in many particular systems, or it can be "tailored" to the needs of a specific application.

An ACS can define one or more of its variables or mappings recursively. In such circumstances, the recursive invocation of the mapping defines a new instance of variables associated with the ACS.

Abstract control structures have three forms in ACS: structures, operations, and functions.

A structure is a relation on the set of mappings, i.e., a set of tuples whose members are sets of ordered pairs. We specify a structure by

```
"STRUCTURE:"  y "=" S "(" x ");"
     declaration...
     definition...
"SYNTAX:"  user defined syntax";"
"END" S ";"

user defined syntax: = connector_1 y_1 "=" S_1 "("x_1")"...
                       connector_n y_n "=" S_n "("x_n")"
```

where x, y are variables or sets of variables whose values are in the same types as the members of the ordered pairs that make up the mappings in the tuples of S;
and S is a structure name;
and $connector_i$ is a user-defined name, possibly empty;
and $y_i = S_i(x_i)$ is an unspecified mapping (see Section 10.0 for use of user-defined syntax).

The unspecified mapping names, used in definition statements within a structure, are nested subscripted names with respect to the root module name. For example,

```
STRUCTURE:  y = F(x);
                 :
           y = F1(g) AND g = F2(x);
           y = F11(h) AND h = F21(g);
                 :
```

A structure is an ACS in which the root module's corresponding function is not specified and in which at least two other members of the same control hierarchy exist as unspecified functions.

The STRUCTURE definitions for the three primitive control structures are defined in Section 12.0. The user-defined syntax can be used in the construction of new structures, operations, or functions.

By an operation, we mean a set of mappings which stand in a particular relation. An operation results, mathematically, from taking particular mappings as the arguments (nodes) of a structure. In AXES, we define a particular operation by means of the following syntax:

```
"OPERATION:"  y "=" L "(" x ");"
     declaration...
     definition...
"END" L ";"
```

where x, y are variables or sets of variables whose values are in the same types as the members of the ordered pairs which are the mappings,
and L is an operation name.

An operation is an ACS in which the root module has a corresponding mapping and in which all the members of the same control hierarchy exist as at least a mapping and at most a function, but not all are functions.

A universal primitive operation is an operation whose arguments can be values of any variable or set of variables. In AXES, universal primitive operations are used with ACS definitions to construct new ACS definitions. The universal primitive operations in AXES are:

The set of CLONE operations

$$(x_1,\ldots x_i) = \text{CLONE}_i(x)$$

here, "x_1" has the value of "x"; "x_2" has the value of "x"...

"x_i" has the value of "x"

The set of IDENTIFY operations

$$(x_{1n_1},\ldots) = \text{IDENTIFY}^{m}_{n_1,\ldots}(x_1,\ldots x_m)$$

where $n_1,\ldots$ is a list of integer values in the range 1 to m and $n_i \neq n_j$;

here, "x_{1_1}" has the value of "x_1",...

"x_{1^n}" has the value of "x_n"

For example,

$$(g,h,i) = \text{IDENTIFY}^{5}_{2,4,5}(a,b,c,d,e)$$

means "g" has the value of "b"

"h" has the value of "d"

"i" has the value of "e"

The set of K operations

$$y = K_{\text{constant}}(x)$$

which maps any value of variable "x" to a constant value.

For example,

$$y = K_{True}(x)$$

means "y" has the value True for any value of variable "x".

AXES introduces a special value of any variable, REJECT. The operation

$$y = K_{REJECT}(x)$$

is used, for example, to construct ACSs for error detection and recovery mechanisms.

By a function we mean a set of mappings which stand in a particular relation for which particular variables have been chosen to represent their inputs and outputs. Whereas structures and operations can be described as purely mathematical constructs, a function is a hybrid, consisting of a mathematical construct, i.e., an operation and a linguistic construct, i.e., an assignment of particular names to inputs and outputs. In AXES, we define a particular function by means of the following syntax:

```
"FUNCTION:"  y "=" F "(" x ");"
    declaration...
    definition...
"END" F ";"
```

where x, y are particular variables or particular sets of variables whose values are in the same types as the members of the ordered pairs which are the mappings,
and F is a function name.

A function is an ACS in which the root module has a corresponding mapping and particular variables and in which all nodes within the module's control hierarchy exist as mappings with particular variables. Note that our use of "function" is slightly

different from what is meant by "function" in mathematics. For the latter notion we use the term "mapping" throughout this report.

9.0 DECLARATIONS

AXES has very simple name rules. If a name is declared outside of any function, operation or structure definition, it is the name of an operation, function, or constant.

A declaration statement has two forms in AXES. A WHERE statement is used to specify a variable as the name of a value of a data type. A PARTITION statement is used to specify nonoverlapping (i.e., mutually exclusive) exhaustive subsets of a set of values.

WHERE

The names used within an ACS are either declared within an ACS definition by a WHERE statement, or are the names used for functions, operations, and types.

In declaration$_1$, x is a variable, y_1,... is a set of variables, T is a constant or variable data type name, and "S" concatenated with T denotes a plural type name.

$$
\text{declaration}_1 := \text{"WHERE"} \left(\begin{array}{l}
\text{x "IS"} \left(\begin{array}{l} \{\text{"A"} \mid \text{"AN"}\} \; [\text{"CONSTANT"}] \; \text{T} \\ \{\text{"A"} \mid \text{"AN"}\} \; \{\text{T}_1 \ldots \mid \text{T}_1\text{"OR"}\ldots\text{T}_n\} \\ \text{"OF SOME TYPE"} \end{array} \right) \\
\text{y}_1\text{,...} \quad \text{"ARE"} \left(\begin{array}{l} [\text{"CONSTANT"}]\ \text{T"S"} \\ \text{T}_1\ldots\text{T"S"} \\ \text{T}_1\text{"S" "OR"}\ldots\text{T}_n\text{"S"} \\ \text{"OF SOME TYPES"} \\ \text{"OF THE SAME TYPE"} \end{array} \right) \text{";"} \\
\text{x "IS"} \left(\begin{array}{l} \text{"("}\text{y}_1\text{","...")"} \\ \text{T} \end{array} \right) \\
\quad \text{"SOME TYPE"}
\end{array} \right)
$$

In the example

```
WHERE x IS A RATIONAL;
WHERE y IS AN INTEGER;
```

x ε RATIONAL.

y ε INTEGER.

In the example

```
WHERE Zero IS A CONSTANT NATURAL;
```

"Zero" represents a particular value of data type NATURAL. Which particular value it represents is determined by the axioms or assertions the example statement might occur with (see Section 15.0).

In the example

```
WHERE x IS AN ARRAY INTEGER;
```

x is an integer member of an ARRAY member.

It is sometimes useful to specify an operator that is capable of operating on data of more than one type. For example, an operation that sums several input arguments could accept both INTEGER and RATIONAL arguments. It is even possible to write operations that will accept arguments of any type. For example, an operation that compared two input arguments for equality could accept arguments of any type. For example,

```
WHERE x IS A NATURAL OR RATIONAL;
```

declares "x" to be a variable capable of being defined to have either NATURAL values or RATIONAL values.

In the example,

```
WHERE x IS OF SOME TYPE;
```

"x" is declared to be a variable whose values are of an unspecified data type.

In the example

WHERE x, y ARE INTEGERS;

x ε INTEGER, y ε INTEGER, i.e., both "x" and "y" represent integers.

In the example,

WHERE x, y ARE SAME TYPE;

"x" and "y" are declared to be variables capable of being defined to have values of the same type, where the type is unspecified.

In the example,

WHERE x IS (x_1,x_2);
WHERE y IS A RATIONAL;

$x = (x_1,x_2)$, i.e., "(x_1,x_2)" is the data structure of "x".

y = RATIONAL, i.e., "y" is a variable whose value is the set of RATIONAL values.

PARTITION

In $declaration_2$, x is a variable or a set of variables enclosed in parentheses, ^jy is variable or set of variables whose values are members of the members of a partition of the set of values of the variables that x represents.

$$declaration_2 ::= \text{"PARTITION OF" } x \text{ "IS" } \begin{Bmatrix} \text{ANY PARTITION} \\ {}^1y \text{ "|" } tv_1 \text{ ","} \ldots {}^iy \text{ "|" } tv_i \text{ ";"} \end{Bmatrix}$$

and

$$\text{true val exp:} \quad \begin{cases} \text{F "("exp}_1\text{")"} \\ \text{F "("exp..."")"} \\ \text{exp F exp} \end{cases}$$

$$tv_i := \begin{cases} \text{true val exp} \\ \text{"("true val exp}_1\text{ ","...true val exp}_i\text{")"} \end{cases}$$

true val exp_i evaluates to the boolean value True, and exp is in terms of x and values of x.

The example

```
PARTITION OF a IS 1a|a > 10,
                  2a|a < 10,
                  3a|a = 10;
```

declares the set $^1a, ^2a, ^3a$ to be a partition of the set of which a is a member.

The example

```
PARTITION OF (a,b) IS 1(a,b)|a>b,
                      2(a,b)|a≤b;
```

declares the set $^1(a,b)$, $^2(a,b)$ to be a partition of the set of which (a,b) is a member.

In the use of a member of a partition, the left superscript can be distributed to each member of a set of variables, e.g., "^{1}(a,b)" can be written "^{1}a", "^{1}b".

10.0 DEFINITIONS

A definition statement defines a particular control level by means of a mapping reference, a primitive definition statement, a user-defined definition statement, or a mapping assertion statement.

In a definition, y,x are variables or sets of variables, and F is a structure, operation, or function.

$$\text{definition}_i: = \begin{cases} y \text{ "=" } F \text{ "(" } x \text{ ")"} \\ \text{primitive definition} \\ \text{user-defined definition} \\ \text{mapping assertion} \end{cases} \text{ ";"}$$

$$\text{Primitive definition: } = \quad \text{definition}_1 \text{ "AND"} \ldots \text{definition}_n \text{ ";"}$$

An example of a primitive definition for the function y = f(h) is

$$y = A(b) \text{ AND } b = C(d) \text{ AND } d = E(h);$$

$$\text{user-defined definition: } = \quad \text{connector}_1 \text{ definition}_1 \ldots \text{connector}_n \text{ definition}_n$$

where a set of connectors is defined in a particular structure definition (see Section 8.0).

An example of a user-defined definition is

$$\text{JOIN } y = f_1(g)$$
$$\text{WITH } g = f_2(a,b) \text{ AND } (a,b) = f_3(x);$$

Section 13 provides more examples of the use of user-defined definitions. Examples of connector definitions are shown in Section 12.0.

mapping assertion: = "WHEREBY" y "=" exp";"

A mapping assertion defines a mapping in terms of operations that have been previously characterized and in terms of bound variables.

If a mapping assertion is used as a definition, the corresponding function has the set of variables referenced in the mapping assertion as input variables. For example,

$y = F_1(a,b)$ might correspond to $F_1(a,b) = a^2 + a/b + 1$.

In this case, we can write

WHEREBY $y = a^2 + a/b + 1$

to define the mapping F_1.

Examples of mapping assertions are:

```
WHEREBY z = g^2 + 1;
WHEREBY (c,d) = (3,4);
WHEREBY z = Sum(Prod(a,b),a);
WHEREBY e = g(k(c));
```

11.0 EXPRESSIONS

Expressions in AXES are similar to expressions of most programming languages.

$$\text{exp} := \left\{ \begin{array}{l} \text{value} \\ \text{x} \\ \text{F(exp)} \\ \text{exp F exp} \\ \text{(exp)} \\ \text{exp,...} \end{array} \right\}$$

where F is an operation or a function name and
x is a variable.

The following is a valid expression:

$$-a_2 + b/(c + f(x)) + 4$$

When operations are used with prefix notation, operator hierarchy and order of evaluation are inherently determined.

For convenience, AXES permits a number of the primitive or auxiliary operations defined on intrinsic data types (c.f. Appendix IV) to be written in terms of the customary prefix or infix symbols. The correspondences between these operations and symbols is given in the following table:

Operation	Symbols
Or, Por	!
And, Pand	&
Not, Pnot, Iopp, Ropp	prefix -
Same, Ident, Equal, ?Equal?, ?Iequal?, ?Requal?	=
?>?, ?I>?, ?R>?	>
Sum, Isum, Rsum	+
Idiff, Rdiff	-
Prod, Iprod, Rprod	*

Operation (con't)	Symbols (con't)
Rdiv	/
Conc	!!

Given these symbolizations, "≥", "<", and "≤" can also be introduced in the usual way as abbreviations for combinations of these symbols. The symbol "+" can also be used as a prefix to indicate the identity mapping, and the symbol "**" can be used to indicate exponents. All symbols in this table are infix symbols, except for the one prefix symbol indicated.

Operator Precedence

The meaning of expressions that contain multiple operators is determined by the relative priority or precedence of the operators and by a property of operators called associativity.

The expression

```
a+b*c
```

for example, means

```
a+(b*c)
```

because "*" has higher precedence than "+". In general, operators of higher precedence take priority over operators of lower precedence. The precedence of AXES operators is as follows:

```
Highest
        ** prefix + -
        * /
        + -
        = < > <= >=
        &
        !
Lowest
```

Operator Associativity

If an expression contains multiple operators of equal precedence, the meaning of the expression is determined by the associativity of the operators. All prefix operators and the "**" infix operator are right-associative, and all other operators are left-associative.

Left-associative operators give priority to other operators of equal precedence to their left, while right-associative operators give priority to operators of equal precedence to their right. For example,

```
a+b+c-d
```

means

```
((a+b)+c)-d
```

while

```
a**b**c
```

means

```
a**(b**c)
```

As in most programming languages, parentheses can be used to clarify the meaning of expressions and to overrule the precedence and associativity of operators.

12.0 PRIMITIVE ABSTRACT CONTROL STRUCTURES AS STRUCTURES

The primitive abstract control structures of HOS can be defined as structures. The structure primitive composition, for example, is the relation that holds among three mappings Cn, Cn_1, Cn_2 if Domain(Cn) = Domain(Cn_2), Range(Cn) = Range(Cn_1), and Domain(Cn_1) = Range(Cn_2), and if, for all x, y such that y = Cn(x), there is a g ε Domain(Cn_1) such that y = Cn_1(g) and g = Cn_2(x). In AXES, we define the particular primitive composition structure by means of the following syntax.

```
STRUCTURE:  y = Cn(x);
WHERE x, y, g ARE OF SOME TYPES;
y = Cn1(g) AND g = Cn2(x);
SYNTAX:  JOIN y = Cn1(g) WITH g = Cn2(x);
END Cn;
```

The structure class partition is the relation that holds between three mappings, Cp, Cp_1, Cp_2 if Domain(Cp) = Domain(Cp_1) X Domain(Cp_2), Range(Cp) = Range(Cp_1) X Range(Cp_2), and, if for all x, y such that y = Cp(x), there is a (y_1,y_2) = y and a (x_1,x_2) = x such that $y_1 = Cp_1(x_1)$ and $y_2 = Cp_2(x_2)$.

In AXES, we define the particular class partition structure by means of the following syntax:

```
STRUCTURE:  (y1,y2) = Cp(x1,x2);
WHERE x1, x2, y1, y2 ARE OF SOME TYPES;
y1 = Cp1(x1) AND y2 = Cp2(x2);
SYNTAX:  INCLUDE y1 = Cp1(x1) ALSO y2 = Cp2(x2);
END Cp;
```

The structure set partition is the relation that holds between three mappings, Sp, Sp_1, Sp_2 if Domain(Sp) = Domain(Sp_1) ∪ Domain(Sp_2) and Domain(Sp_1) ∩ Domain(Sp_2) = ∅, and Range(Sp) = Range(Sp_1) ∪ Range(Sp_2) and if, for all x, y such that y = Sp(x), there is either a y = Sp(^{1}x) or a y = Sp(^{2}x).

In AXES we define the particular set partition structure by means of the following syntax:

STRUCTURE: y = Sp(x);

WHERE x, y ARE OF SOME TYPES;

^{1}y = Sp_1(^{1}x) AND ^{2}y = Sp_2(^{2}x);

PARTITION OF (x,y) IS ANY PARTITION;

SYNTAX: EITHER ^{1}y = Sp_1(^{1}x) OTHERWISE ^{2}y = Sp_2(^{2}x);

END Sp;

13.0 THE USE OF STRUCTURES, OPERATIONS AND FUNCTIONS

In the following examples, the data types referenced are intrinsic data types of AXES (Appendix IV) unless otherwise indicated.

An example of an operation using the Cn structure is:

```
OPERATION:  y = Transform (a,b)
WHERE a,b ARE RATIONALS;
WHERE y,g ARE INTEGERS;
JOIN y = A(g) WITH g = B(a,b);
END Transform;
```

In this example, the structure input variable has been replaced by a list of variables.

A function can include the use of structures and operations. For example, function G uses the Cn structure and the Transform operation.

```
FUNCTION:  c = G(d);
WHERE h IS A RATIONAL;
WHERE c1,c2,h1,h2,d ARE INTEGERS;
WHERE c = (c1,c2);
WHERE h = (h1,h2);
JOIN c = R(h) WITH h = Transform(d);
INCLUDE c1 = T(h1) ALSO c2 = W(h2);
END G;
```

An example of a function using the Sp structure is:

```
FUNCTION:  y = Decide (x);
WHERE x,y ARE INTEGERS;
EITHER y = A(1x) OTHERWISE y = B(2x);
PARTITION OF x is 1x|x<10, 2x|x≥10;
END Decide;
```

In function S, function Calculate begins at time b and completes at time a. Here the execution time span of Calculate is defined by function Clock. We assume there, an extrinsic data type, Timeslot, where x and y are Timeslots whose values are integers and an extrinsic data type, TIME.

FUNCTION: $(a,y_a) = S(b,x_b)$;
WHERE y,x ARE TIMESLOT INTEGERS;
WHERE a,b are TIMES;
INCLUDE a = Clock(b) ALSO y_a = Calculate(x_b);
END S;

A complete system specification is defined as a structure of functions whose lowest level functions are primitive operations on the data types represented by the variables of those lowest level functions. An example of a complete system definition is system F. The lowest level functions of system F_1 (i.e., F_1 and F_2) are described in terms of primitive operations on RATIONALS. The intent of system F is to perform two independent functions.

FUNCTION: $(y_1,y_2) = F(x_1,x_2)$;
WHERE y_1,y_2,x_1,x_2 ARE RATIONALS;
INCLUDE $y_1 = F_1(x_1)$ ALSO $y_2 = F_2(x_2)$;
END F;

FUNCTION: $y_1 = F_1(x_1)$;
WHERE y_1,x_1,g ARE RATIONALS;
JOIN WHEREBY $y_1 = g+1$ WITH WHEREBY $g = x_1^2+3y_1$;;
END F_1;

FUNCTION: $y_2 = F_2(x_2)$;

WHERE y_2, x_2 ARE RATIONALS;

EITHER WHEREBY $y_2 = a^2$, OTHERWISE WHEREBY $y_2 = b^3$;

PARTITION OF x_2 is $a|x_2<10$, $b|x_2\geq 10$;

END F_2;

The control map for System F is shown in Figure 13.1.

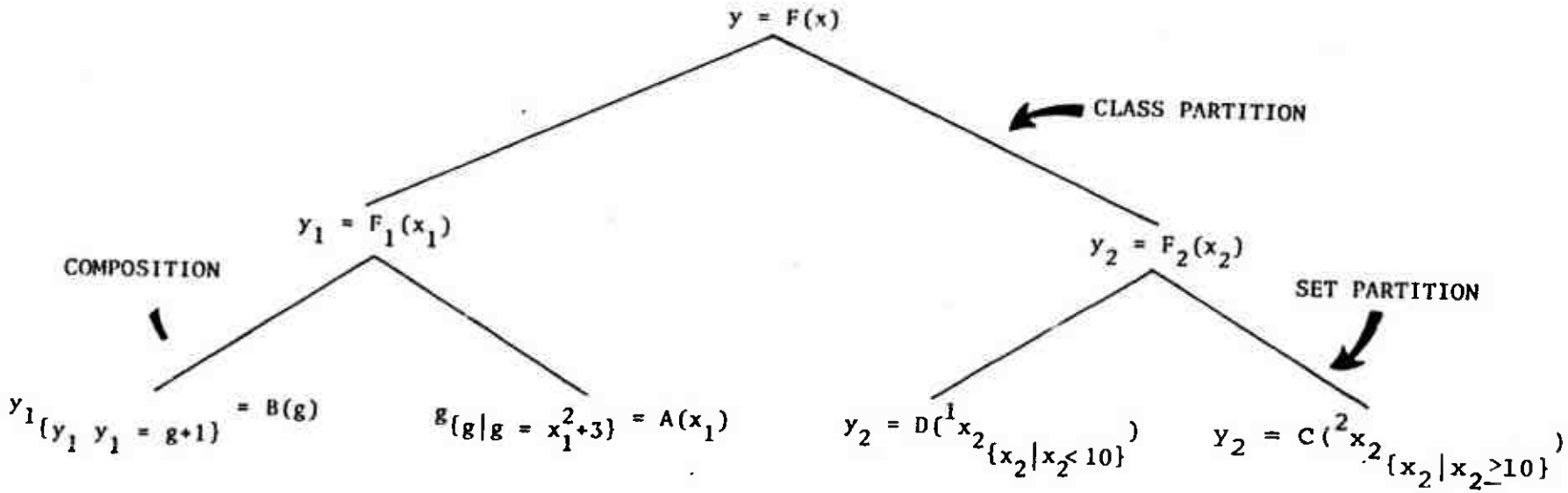


FIGURE 13.1: Decomposition of System F

14.0 FUNCTION FAILURE

Failure of a function means that none of its outputs are defined.

If a function fails, the failure could propagate back up to the function that referenced it, etc., all the way back to the top of the system, causing the system to fail.

To permit orderly recovery from errors, AXES provides a structure in which provision for failure can be expressed.

```
STRUCTURE:  w = FAIL(x,y);
WHERE x,y,w ARE OF SOME TYPES;
JOIN w = Fail1(w,1y) WITH (w,y) Fail2(x,y);
          1  1        1 1
EITHER (1w,1y) = Fail1(1(x,y)) OTHERWISE
        1  1       12
WHEREBY 2w = REJECT, 2y = 2y;
        1            1
PARTITION OF 1(x,y,w,y) IS
               1 1

  1(x,y,w,y)|(x NOT= REJECT, w NOT= REJECT),
    1 1                      1

  2(x,y,w,y)|(x=REJECT,w=REJECT);
    1 1              1

JOIN (1w,1y) = Fail1  (1x,1y,1y) WITH
      1  1       12   1  2  3

    (1y,1y) = CLONE2(1y) AND 1x = CLONE1(1x);
      2  3                    1

INCLUDE 1w = Fail1   (1x,1y) ALSO 1y = CLONE1(1y);
        1       12    1  2        1           3

INCLUDE 1w = Fail1 (1w,1y) ALSO 2w = IDENTIFY21((2w,2y);
                11  1  1                        1  1

PARTITION OF (w,y,w) IS
               1 1

  1(w,y,w)|w = REJECT,
    1 1    1

  2(w,y,w)|w NOT= REJECT;
    1 1
```

JOIN $^{1}_{1}w_{1}$ = $Fail_{1_1}$ ($^{1}_{2}y$) WITH $^{1}_{2}y$ = $IDENTIFY^{2}_{2}(^{1}_{1}w, ^{1}_{1}y)$;

SYNTAX: $^{1}_{1}w_{1}$ = $Fail_{1_1}$ (x,y) FAILURE $^{1}_{1}w_{1}$ = $Fail_{1_1}$ (y);
1_2

END Fail;

Using the Fail structure, a function definition statement such as

z = F(x,y,a) FAILURE z = G(y,a);

implies if F(x,y,a) fails, G(y,a) will be used to define the value for z. If G(y,a) might fail, we can write:

z = F(x,y,a) FAILURE z = G(y,a) FAILURE z = H(a);

If H(a) might fail, the failure propagates back to the controller function which either uses a Fail structure of its own or further propagates the error back up the system.

15.0 DATA TYPES

Abstract Types

The meaning of a type of value is determined by the set of operations that can be performed on the values of that type. For example, what it means to be an integer is determined by the set of operations that can be performed on integers.

Most programming languages define a limited set of data types and a set of operations on those data types. More advanced languages allow the programmer to define new data types in terms of existing or previously defined data types. Because these definitions define the new data types in terms of base types, however, the newly defined types usually exhibit the properties of their base types. For example, if the type department_numbers is defined in terms of the type INTEGER, department_numbers are likely to be permitted as operands of arithmetic operators.

In AXES, new data types can be defined simply in terms of the operations that are to be performed on the data (GUT75) (Appendixes III and IV). To specify a data type in AXES, we use the following syntax:

```
"DATA TYPE:"  name ";"
"PRIMITIVE OPERATIONS;"
    primitive operations...
"AXIOMS;"
    declaration...
    assertion (about a type)...
"END" name";"
```

where

(1) name is the abstract data type name.

(2) the primitive operations are not defined in terms of other operations, but in terms of each other. As with all operations, the name of each primitive operation is known throughout the

system specification, and each primitive operation can be referenced from within any ACS and from within other data type definitions.

primitive operation:= typename_i "=" P_i "(" typename_j,..."); "

where typename_k is a data type name in lower-case characters and k is an integer, possibly empty, and P_i is a primitive operation name.

(3) An assertion (about a type) in AXES is a true statement about the equality of two ACSs in which all the nodes are operations. Each ACS is defined in terms of primitive operations of the data type of interest or of previously characterized primitive operations on another data type. The arguments of a mapping can be values of a type as an alternate notation for the $K_{CONSTANT}$ operation (Section 8.0) or bound variables. The set of assertions (about a type) completely characterize the type of interest (Appendix III).

$$\text{assertion (about a type): } = \left\{\begin{matrix}\text{definition}_1 \\ F"("\text{exp}_1")"\end{matrix}\right\} \text{ "=" } \left\{\begin{matrix}\text{definition}_2 \\ \text{exp}_2\end{matrix}\right\} ";"$$

where exp_i is an exp in terms of previcusly characterized or primitive operations, variables that represent values of previously characterized data types or the type of interest, and values of previously characterized data types or the type of interest; F is an operation name; definition_i is in terms of the same objects as exp_i; and either F or at least one of the operations of exp_i are primitive.

An example of a data type specification is data type STACK.

```
DATA TYPE:  STACK;
PRIMITIVE OPERATIONS:

stack1 = Push(stack2,integer1);
stack1 = Pop(stack2);
integer1 = Top(stack1);

AXIOMS:
     WHERE Newstack IS A CONSTANT STACK;
     WHERE s IS A STACK;
     WHERE i IS AN INTEGER:

Top(Newstack) = REJECT;
Top(Push(s,i)) = i;
Pop(Newstack) = REJECT;
Pop(Push(s,i)) = s;
END STACK;
```

The entire set of statements constitutes the definition of the type STACK. The first line and last line give the name of the abstract type, the lines between "DATA TYPE" and "AXIOMS" are the complete set of primitive operations that receive or define values of the type STACK.

The names STACK and INTEGER that appear within the primitive operations are the names of types (INTEGER is a previously defined type; STACK is the type we are defining). Only type names and primitive-operation names appear within the primitive-operation list. The lines following the word AXIOMS are axioms, or assertions, about the behavior of the primitive operations. Any actual implementation of the primitive operations must satisfy these axioms. If it does not, the implementation is invalid and does not meet the specification. Note that this method of specifying a STACK does not bias the final implementation toward the use of any particular mechanism such as linked list, array, etc.

The names "i" and "s" that appear within the axioms represent any value of any variable of the required type. Because "i" is declared to be an INTEGER variable, it represents values of INTEGER variables. Similarly,"s"represents values for variables of type STACK. On the other hand, "Newstack" represents a particular value that "s" can represent.

Within a set of axioms, a variable of a given data type can be used only in contexts that require that same data type.

Each axiom must be true for all possible values of the variables described. For example, the second axiom means that for any STACK s, and any INTEGER i, the resulting values of the nested operations Top(Push(Stack,i)) must be equal to the value i. (Pushing i onto s and then taking the top item off of the STACK produced by the Push, must yield value i.)

Intrinsic Types

Because certain data types are common to a wide range of system specifications, they are predefined by AXES. This means that the system designer does not have to define these types. For AXES, Appendices IV and V contain the intrinsic data type definitions.

$$\text{intrinsic types: } = \left\langle \begin{array}{l} \text{boolean} \\ \text{natural} \\ \text{integer} \\ \text{rational} \\ \text{property (of T)} \\ \text{set (of T)} \\ \text{line} \end{array} \right\rangle \qquad \text{value: } = \left\langle \begin{array}{l} \text{boolean value} \\ \text{natural value} \\ \text{integer value} \\ \text{rational value} \\ \text{property (of T)value} \\ \text{set (of T) value} \\ \text{line value} \\ \text{extrinsic value} \end{array} \right\rangle$$

boolean value: = { True | False }

natural value: = { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 } ...

integer value: = {+ | -} natural

rational value: = { integer$_1$. | integer$_1$.integer$_2$ } ["E" integer]

property (of T) value: : = "PROPERTY OF" t "IN" T" | "true val exp$_1$

set (of T) value: = { {value$_1$,...} | "SET OF" t "IN" T "|" true val exp$_1$ }

line value: = 'any finite string of symbols possibly empty'

Extrinsic data type values are defined as "'CONSTANT' T" using a declaration$_1$ statement (Section 9.0).

16.0 DERIVED OPERATIONS

In AXES, we specify the behavior of an operation without specifying its decomposition by writing it as a derived operation.

The meaning of a derived operation can be determined in terms of previously characterized operations (i.e., derived from primitive operations of a type).

In AXES, we specify a derived operation implicitly by means of assertions (about an operation) that describes the behavior of the operation with respect to other already-defined operations. The existence of a derived operation of some types must be provable mathematically from the existence of the primitive operations and the axioms of those types. To specify a derived operation in AXES, we use the following syntax.

```
"DERIVED OPERATION:"  y "=" D "(" x ");"
        declaration...
        assertion(about D)...
"END" D ";"
```

where x,y are variables or sets of variables and D is a derived operation name.

$$\text{assertion(about D)}: = \begin{Bmatrix} \text{definition}_1 \\ F_1\text{"("exp}_1\text{")"} \end{Bmatrix} \text{ "=" } \begin{Bmatrix} \text{definition}_2 \\ F_2\text{"("exp}_2\text{")"} \end{Bmatrix} \text{ ";"}$$

where F_i is D or an operation of the types D is derived from; exp_1 is an exp in terms of the derived operation, D, or operations of the types D is derived from and values of the types D is derived from; $definition_i$ is in terms of the derived operation D or operations of the types D is derived from and values of the type D is derived from. For example, a derived operation taken from Appendix IV is

DERIVED OPERATION: $integer_3$=IGCD($integer_1$,$integer_2$);

WHERE i_1,i_2 ARE INTEGERS;

Abs(IGCD(i_1,i_2)) = GCD(Abs(i_2,i_2));

Sign(IGCD(i_1,i_2)) = True;

END IGCD;

17.0 RECURSION VS. ITERATION

Because AXES does not permit multiple definitions of a given variable, the common control statements of programming languages have no meaning in AXES. For example, a statement of the form

```
WHILE a<b DO ... END;
```

has no meaning in AXES because a<b always has the same value. (The values of a and b cannot change.)

Traditional control statements, such as WHILE, are normally used to express functions as iterative algorithms. In AXES, these iterative algorithms must either be expressed using control structure definitions or written as recursive functions.

The following text shows a simple iterative algorithm written in PL/1:

```
DO WHILE(i,n);
a(i) = F(b(i));
i = g(i);
END;
```

In AXES, the same problem is expressed using functions without giving more than one definition of any variable.

This PL/1 formulation cannot be tested for interface correctness. For example, we do not know if a(i) is defined more than once, or if a(i) is ever defined. This uncertainty could have serious consequences on system implementation; but we can avoid the

problem altogether in AXES by formulating the function differently. Suppose we wished to establish the values that "a" represents where a = $(a_1,...a_i)$.

FUNCTION; a = H(b);

WHERE a = (a_1,a_2); WHERE b = (b_1,b_2);

INCLUDE $a_1 = F_1(b_1)$ ALSO $a_2 = G(b_2)$;

EITHER $a_2 = K_{REJECT}({}^1b_2)$ OTHERWISE

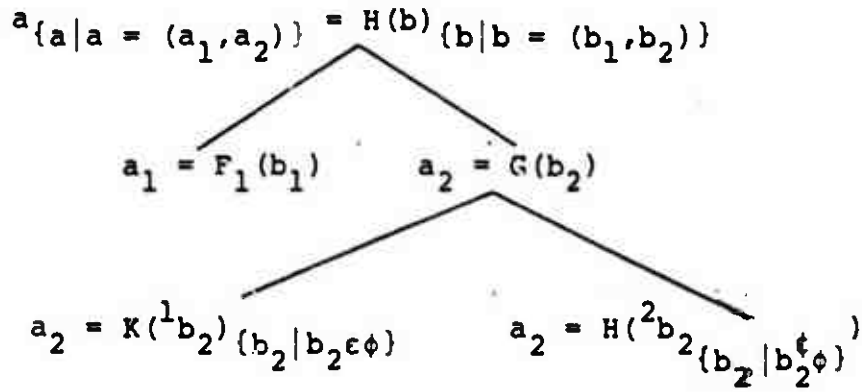
$a_2 = H({}^2b_2)$

PARTITION OF b_2 IS ${}^1b_2|b_2$ = REJECT,

${}^2b_2|b_2$ NOT = REJECT;

END H;

The control map for function H is



In the AXES formulation, any size data structure is inherent in the specification, however, a and b possess the same structure. If "a" and "b" are defined as variables and H is a primitive operation on data type array, then the above specification can be expressed simply as "a = H(b)".

18.0 CONCLUDING REMARKS

In this report we have been careful to differentiate between the name of a object and the object, itself.

In AXES, a name (variable) always represents no more than one object. The objects are members of a hierarchy whose relationship is that of control. AXES syntax, AXES abstract control structures, AXES abstract data types, and AXES systems are all based on the methodology of HOS. In this version of AXES, we have concentrated on a syntax which will provide a basis for a system to be explicitly defined in such a way or to be automatically analyzed for interface correctness.

In the future we hope to provide more building blocks (i.e. user-defined abstract control structures and abstract data types) in order to facilitate further the communication between users in the process of specifying particular systems and systems in general.

BIBLIOGRAPHY

HAM76a Hamilton, M. and Zeldin, S., "The Foundations for AXES: A Specification Language Based on Completeness of Control", Rev., Doc. R-964, The Charles Stark Draper Laboratory, Inc., March 1976.

HAM76b _____, and _____, "Higher Order Software: A Methodology for Defining Software", IEEE Transactions in Software Engineering, Vol. SE-2, No. 1, March 1976.

HAM76c _____, and _____, "Integrated Software Development System/Higher Order Software Conceptual Description", Version 1, Higher Order Software, Inc. Cambridge, MA, November 1976.

GUT75 Guttag, J., "The Specification and Application to Programming Abstract Data Types", Univ. of Toronto Technical Report, CSRG-59, September 1975.

LAN52 Laning, H., "MAC Language", The Charles Stark Draper Laboratory, 1952.

LIC74 Lickly, D.J., et al., "HAL/S Language Specification", Intermetrics, Inc., Cambridge, MA, 1974.

APPENDIX I

PRELIMINARIES OF HOS

APPENDIX I
PRELIMINARIES OF HOS

Trees and Functions*

Using the HOS approach, software systems can be developed with the aid of simple mathematical concepts and a set of software engineering axioms. In this Appendix, the required mathematical concepts are described.

The two mathematical concepts required in order to describe HOS are the tree and the function. The tree is a structure comprised of a finite number of nodes which are connected by branches as shown in Figure AI-1.

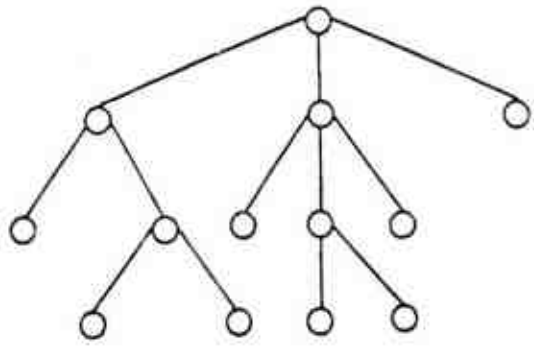


Figure AI-1
An Example of a Tree Structure

A branch may be interpreted as entering a node (from above the node) or leaving a node (from below). The unique node at the top of the tree that has no branches entering it is called the root of the tree. A node that has no branches leaving it is called a leaf of the tree. It should be noted that all nodes other than the root have exactly one entering branch.

A root is considered to be at level 0 of the tree (see Figure AI-2). As one starts at the root and traverses a path to a leaf,

*Excerpted from Hamilton, M. and Zeldin, S, "Integrated Software Development System/Higher Order Software Conceptual Description", Version 1, Higher Order Software, Inc., Cambridge, MA, Nov. 1976.

each successive node defines the next level of the tree.

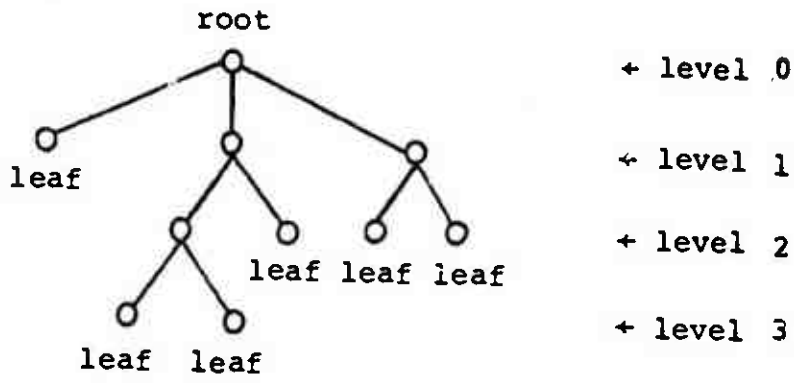


Figure AI-2
Tree Levels

If a branch leaves node A (Figure AI-3) and enters node B, then node A is the parent of node B, and node B is an offspring of node A. (In Figure AI-3 node C is also an offspring of node A.)

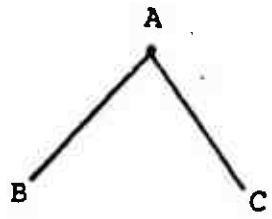


Figure AI-3
Parent-Offspring Relationship

A nodal family is a particular parent node and all of its offspring (see Figure AI-4).

If there exists a sequence of nodes $n_1, n_2, \ldots, n_k$, such that for every i, n_{i+1} is an offspring of n_i, then each n_{i+1} is a descendant of n_1. A particular parent node of the tree together with all of its descendants and connecting branches is the subtree defined by the given parent.

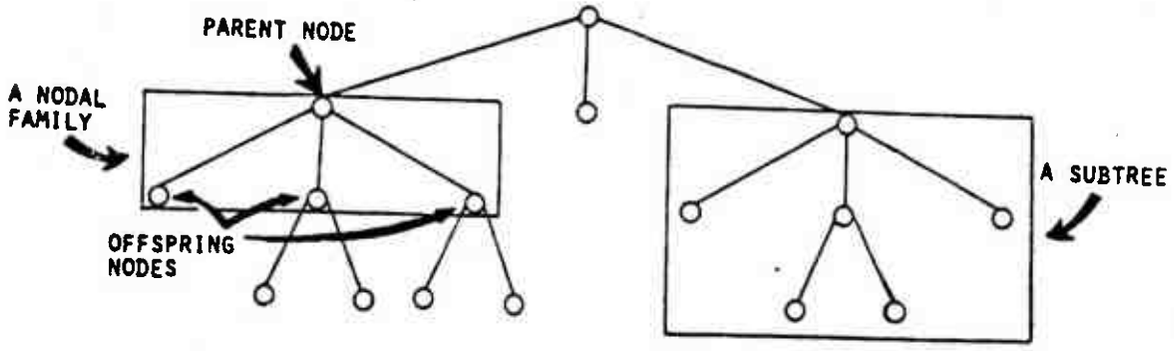


Figure AI-4
Tree Substructures

If α and β are set elements (from either the same or different sets), then "(α,β)" denotes the ordered pair consisting of α and β in that order. (Thus, the ordered pair (α,β) is not the same as (β,α) except for the case where α and β are the same elements.)

If two sets, X and Y, are given, and "x" and "y" represent arbitrary elements of X and Y, respectively (i.e., "x" and "y" are variables), then any set of ordered pairs of the form (x,y) is a relation between X and Y. For example, if X = {1,2,3,4,5,6} and Y = {m, s,e,w}, then one possible relation between X and y is R = {(4,m), (3,s), (4,w)}.

The set of left elements of the relation is called the domain, and the set of right elements, the range. In the above example, the domain is {3,4}, and the range is {m,s,w}.

A relation is a function when each element of the domain has only one corresponding range element. If f is a relation between X and Y, and f is also a function, then we say that "f is a function from X into Y" (usually written y = f(x)). An example of a function is

$$f = \{(1,m), (2,s), (4,m), (6,e)\}$$

as illustrated in Figure AI-5.

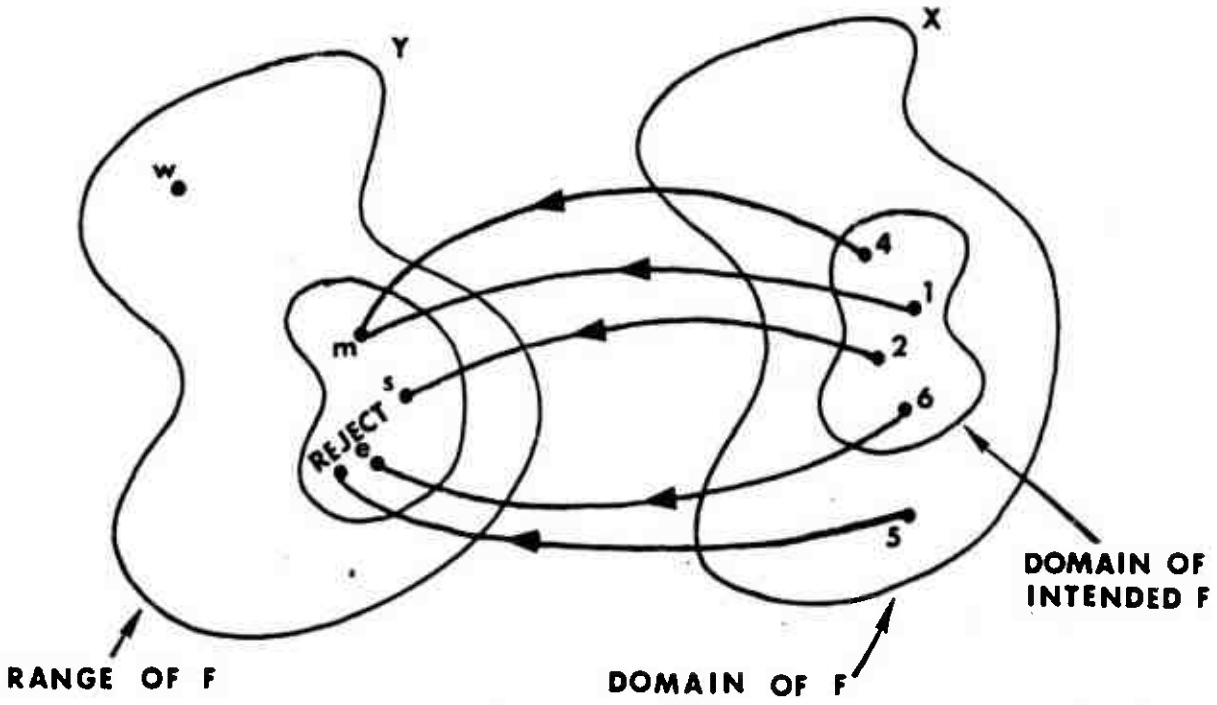


Figure AI-5
Illustration of a Function from X into Y

In the sections that follow, the variable that represents the domain elements is referred to as the input variable, and the variable that represents the range elements is referred to as the output variable. Individual domain and range elements may be called inputs and outputs, respectively.

Modules and Nodal Families

In HOS, the decomposition process for a system results in a tree structure. At the start of the decomposition process, the entire system is represented by the root of the tree which, hopefully, represents the requirements for the system. This description, however, has many implicit (hidden) requirements. In order to explicitly arrive at the complete description of the requirements of the system, the root is decomposed by replacing it by a nodal family, which represents the decomposition of the root. This decomposition process, that of replacing a function by its nodal family, can be continued until the entire system has been explicitly specified to whatever detail is required or desired.

It may turn out that during the decomposition process, a requirement is shown to be erroneous or missing. In such a case, an iteration of the system description is required.

The parent node of the nodal family controls its offspring. When referring to this control relationship, the parent node will be called a module, and its offspring will be called functions. The offspring of the nodal family are the functions required to perform the module's corresponding function (MCF) (i.e., the function that the nodal family replaces.

The resulting tree represents the system where the leaves represent, in an abstract machine sense, the machine "instructions" that are to be actually performed; the intermediate nodes represent control with respect to the performance of these leaves. It can be shown that the HOS axioms provide rules for the way that a nodal family can be constructed (HAM76).

HIGHER ORDER SOFTWARE, INC. · 843 MASSACHUSETTS AVENUE · CAMBRIDGE, MASSACHUSETTS 02139 · (617) 661-8900

DEFINITION: Invocation *provides for the ability to perform a function.*

AXIOM 1: A given module controls the invocation of the set of functions on its immediate, and only its immediate, lower level.

DEFINITION: Responsibility *provides for the ability of a module to produce correct output values.*

AXIOM 2: A given module controls the responsibility for elements of only its own output space.

DEFINITION: *An* output access right *provides for the ability to locate a variable, and once located, the ability to place a value in the located variable.*

AXIOM 3: A given module controls the output access rights to each set of variables whose values define the elements of the output space for each immediate and only each immediate, lower level function.

DEFINITION: *An* input access right *provides for the ability to locate a variable, and once located, the ability to reference the value of that variable.*

AXIOM 4: A given module controls the input access rights to each set of variables whose values define the elements of the input space for each immediate, and only each immediate, lower level function.

DEFINITION: Rejection *provides for the ability to recognize the improper input element in that if a given input element is not acceptable, null output is produced.*

AXIOM 5: A given module controls the rejection of invalid elements of its own, and only its own, input set.

DEFINITION: Ordering *provides for the ability to establish a relation in a set of functions so that any two function elements are comparable in that one of said elements precedes the other said element.*

AXIOM 6: A given module controls the ordering of each tree for the immediate, and only the immediate, lower level.

HIGHER ORDER SOFTWARE, INC. · 843 MASSACHUSETTS AVENUE · CAMBRIDGE, MASSACHUSETTS 02139 · (617) 661-8900

APPENDIX I BIBLIOGRAPHY

HAM76 Hamilton, M., and Zeldin, S., "The Foundations for AXES: A Specification Language Based on Completeness of Control", Rev., Doc. R-964, The Charles Stark Draper Laboratory, Inc., March 1976.

APPENDIX II

PROPERTIES OF THE PRIMITIVE CONTROL STRUCTURES

PROPERTIES OF THE PRIMITIVE CONTROL STRUCTURES*

FUNCTIONAL DECOMPOSITION

While a function can be decomposed in many ways, the HOS axioms provide rules for the construction of nodal families (i.e., the decomposition of a function). From the axioms, three primitive control structures are derived which are used for functional decomposition (HAM76b). These control structures are composition, set partition, and class partition.

Composition is illustrated in Figure AII-1. In order to perform $f_1(x)$, the function f_2 must first be applied to x which results in output z. z then becomes an input to f_3 which produces the desired range element of the overall function f_1.

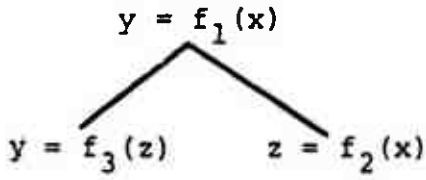


Figure AII-1: An Example of Composition

It is important to observe the following characteristics of composition (characteristics are explained with respect to the example in Figure AII-1):

(1) One and only one offspring (specifically f_2 in this example) receives access rights to the input data, x, from module f_1.

(2) One and only one offspring (specifically f_3 in this example) has access rights to deliver the output data, y, for module f_1.

---

* Excerpted from Hamilton, M. and Zeldin, S., "Integrated Software Development System/Higher Order Software Conceptual Description", Version 1, Higher Order Software, Inc., Cambridge, MA, Nov. 1976.

(3) All other input and output data that will be produced by offspring controlled by f_1 will reside in <u>local</u> variables (specifically "z" in this example). Local variable, "z", provides communication between the offspring, f_2 and f_3.

(4) Every offspring is specified to be invoked once and only once in each process of performing the parent modules corresponding function (MCF).

(5) Every local variable must exist both as an input variable for one and only one function and as an output variable for one and only one different function on the same level.

Additional examples of composition are given in Figure AII-2 and Figure AII-3.

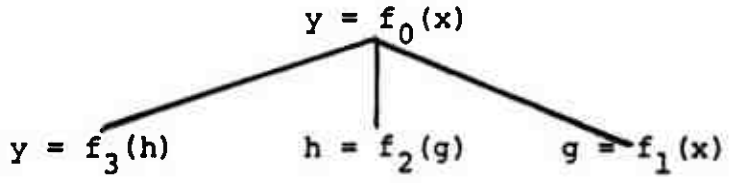


Figure AII-2: Composition with Three Functions on One Level

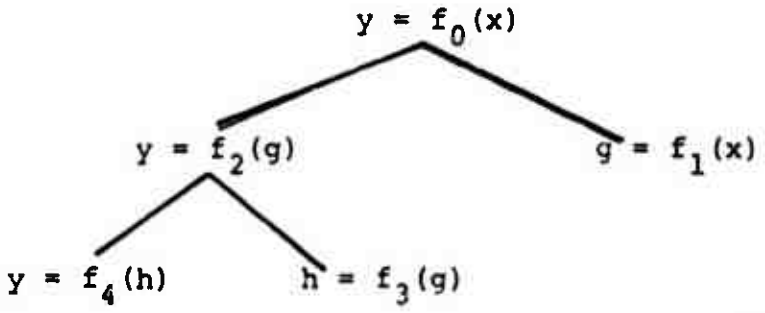


Figure AII-3: Multilevel Composition

Set partition, which involves partitioning of the domain, is illustrated in Figure AII-4. In this example, the set which comprises the domain is partitioned* into two subsets. For set partition, only one of the offspring will be invoked for each performance of the MCF at f_1 (the determination being based on the value of "x" received) and that offspring will produce the required range element for its parent module when it is performing.

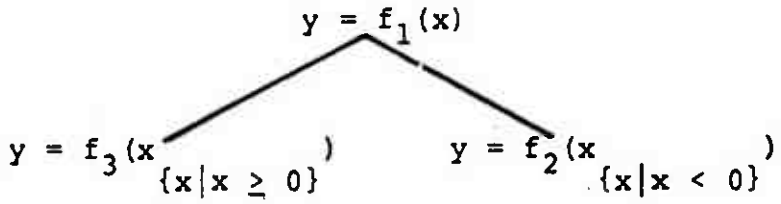


Figure AII-4: An Example of Set Partition

The following characteristics with respect to set partition should be observed:

(1) Each offspring of the module at f_1 is granted permission to produce output values of "y".

(2) All offspring of the module at f_1 are granted permission to receive input values from the variable "x".

(3) Only one offspring is specified to be invoked per input value received for each process of performing its MCF, i.e., only one offspring has a state change for a given state change of the parent module.

(4) The values represented by the input variables of an offspring's function comprise a proper subset of the domain of the function of the parent module.

(5) There is no communication between offspring.

*Partitioning implies the subdivision of the original set into non-overlapping (i.e., mutually exclusive) subsets.

Alternative approaches to the set partition illustrated in Figure AII-4 are presented in Figures AII-5 and AII-6.

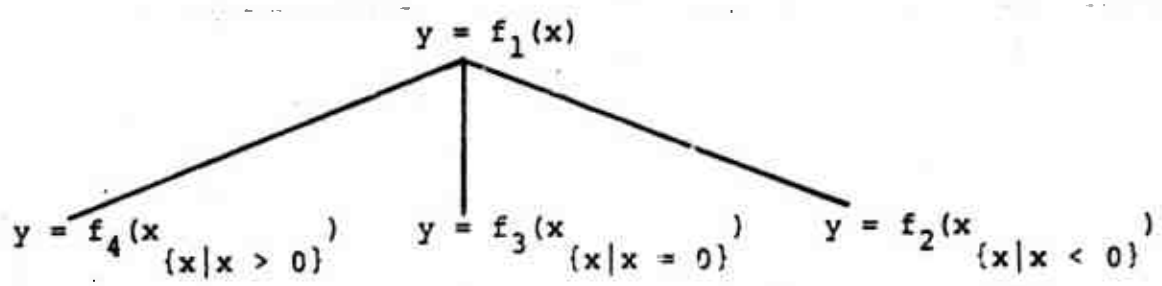


Figure AII-5: Set Partition with Three Functions on One Level

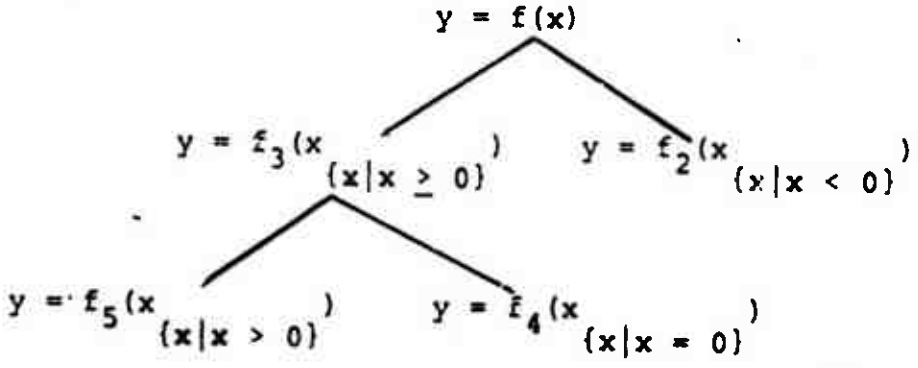


Figure AII-6: Multilevel Set Partition

Class partition is illustrated in Figure AII-7. While set partition involves partition of the domain into subsets, class partition involves partition of the domain variables into classes and the partition of the range variables into classes. In the example, it is assumed that the domain variable has an associated data structure comprised of two parts, "x_1" and "x_2". Likewise, the range variable has an associated data structure with the same

number of classes as the domain's data structure. (As an example of such a structure, consider the domain to be the complex numbers; the range to be polar coordinates. Then, for a given value of the domain variable (i.e., a given complex number), "x_1" would represent its real part and "x_2" its imaginary part. Consequently, the variable is partitioned into two separate classes, "x_1" and "x_2", such that elements associated with "x_1" are the input elements that one offspring can access and the elements associated with "x_2" are the input elements that the other offspring can access. The range structure is partitioned in a similar manner.

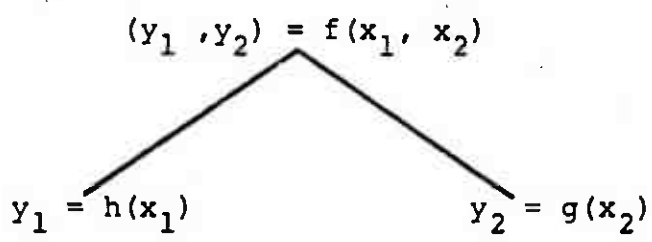


Figure AII-7: An Example of Class Partition

The following characteristics with respect to class partition should be observed.

(1) All offspring of the module at f are granted permission to receive input values taken from a partitioned variable in the set of the parent MCF domain variables, such that each offspring's set of input variables are non-overlapping and all the offspring input variables collectively represent only its parent's MCF input variables.

(2) All offspring of the module at f are granted permission to produce output values for a partitioned variable in the set of the parent MCF range variables, such that each offspring's set of output variables are non-overlapping and all the offspring's output variables collectively represent the parent MCF output variables.

(3) Each offspring is specified to be invoked such that for each change in state of its parent, all offspring undergo a state change.

(4) There is no communication between offspring.

APPENDIX III

ALGEBRAIC SPECIFICATION OF ABSTRACT DATA TYPES

by S. Cushing

APPENDIX III

ALGEBRAIC SPECIFICATION OF ABSTRACT DATA TYPES

An algebra is an ordered pair $[\Sigma,\omega]$, where Σ is a non-empty class of non-empty sets, and ω is a non-empty class of operations on the grandmembers (i.e., members of members) of Σ. The members of Σ are called categories[1] of the algebra, and the members of ω are called the primitive operations of the algebra. A particular algebra can be specified by giving a category specification, an operational[2] specification, and an axiom specification. A category specification lists or defines the members of Σ. An operational specification gives the domains and ranges of the members of ω as Cartesian products of the members of Σ. An axiom specification is a non-empty set of formal statements that characterize the interactive behavior of the members of ω and the grandmembers of Σ. Algebras can be classified according to the constraints that we choose to put on one or more of their category, operational, or axiom specifications.

An algebra $[\Sigma,\omega]$ is said to be homogeneous, if Σ contains exactly one non-empty member. The most familiar kind of homogeneous algebra is probably the group. A non-empty set G is said to be a group with respect to a binary operation Mult, called the group multiplication, defined on G, if (1) G is closed under Mult[3], (2) Mult is associative, (3) there is an element in G that is neutral with respect to Mult, and (4) every element of G has an inverse. which produces the neutral element under Mult (c.f. (FUN74)). We can specify a group G formally as a homogeneous algebra as follows:

---

[1] Birkhoff (BIR70) and Guttag (GUT75) call these phyla. The categories or phyla of a type algebra, which we will consider later, we will call types.

[2] Guttag (GUT75) calls this the syntactic specification, but this term is somewhat misleading.

[3] The uniqueness of the image under Mult is also required, but this is guaranteed by specifying Mult as a mathematical function, i.e., mapping, as in (I,2).

HIGHER ORDER SOFTWARE, INC. · 843 MASSACHUSETTS AVENUE · CAMBRIDGE, MASSACHUSETTS 02139 · (617) 661-8900

(I) Group $G = [\Sigma, \omega]$:

(1) Σ: G, Neut ε G

(2) ω: Mult: G x G → G

Inv: G → G

(3) Axioms: 1. $Mult(g_1, Mult(g_2, g_3)) = Mult(Mult(g_1, g_2), g_3)$

2. Mult(Neut,g) = g

3. Mult(g,Neut) = g

4. Mult(g,Inv(g)) = Neut

In this example, (1) is the category specification of the group G, (2) is its operational specification, and (3) is its axiom specification.

The category specification (1) says that the algebra G contains exactly one set G and that there is an element Neut in G. The operational specification (2) says that the algebra G contains two primitive operations. The first of these (Mult) produces a member of G from an ordered pair of members of G, and the second (Inv) produces a member of G from a member of G.

The axiom specification (3) specifies the interactive behavior of the members of the set specified in (1) and the primitive operations specified in (2). Every axiom should be interpreted as being universally quantified over each of its free variables. Axiom 1 says that Mult is an associative operation. Axioms 2 and 3 taken together say that Mult has no effect, if Neut is one of its arguments. These axioms are often combined as a single axiom of the form:

(4) Mult (Neut,g) = Mult (g,Neut) = g,

but we have given them as separate axioms to ensure that every axiom uniformly contains exactly one equality symbol. Axiom 4 says that Mult maps every member of G and its inverse onto the

neutral element. Together Axioms 1-4 provide the primitive operations in ω with the properties required to make the set in Σ a group.

Other familiar examples of homogeneous algebras are the modules, rings, and fields. A group is said to be a module,[4] also called an Abelian group, with respect to its group operation, if that operation is commutative[5]. A ring is a non-empty set on which two operations, Sum and Mult, are defined such that it is a module with respect to Sum, and Mult is associative and distributive in both directions over Sum. A field is a ring in which Mult is commutative and every element other than the neutral element with respect to Sum has an inverse with respect to Mult.

We can formalize these notions very easily in terms of the framework being developed here. To get a module we simply add the axiom:

$$(5)\ \text{Mult}\ (g_1,g_2) = \text{Mult}\ (g_2,g_1)$$

To get a ring, we first replace "Mult" throughout (I) and Axiom 5 with "Sum", "Neut" with "Zero", and "Inv" with "Opp", meaning opposite. These names are changed simply to bring them more in line with our intuitive interpretation of Sum as a kind of addition. We alsc replace "*G*" with "*R*", "G" with "R", and "g" with "r". Then we put "Mult" back into (2), we put Axiom 1 back into (3), and we add the two axioms

$$\text{Mult}\ (r_1,\text{Sum}(r_2,r_3)) = \text{Sum}\ '\text{Mult}(r_1,r_2),\ \text{Mult}\ (r_1,r_3))$$

$$\text{Mult}\ (\text{Sum}(r_1,r_2),r_3) = \text{Sum}\ (\text{Mult}(r_1,r_3),\ \text{Mult}\ (r_2,r_3)).$$

---

[4] It should be emphasized that this mathematical use of the term "module" is entirely unrelated to what is meant by the term "module" in systems analysis, particularly in HOS. We use it here only as an example and will not use it outside of this Appendix.

[5] Modules are customarily written with additive operations, like Sum, but, strictly speaking, any Abelian group is a module, since the name of any particular operation is arbitrary. See Note 4.

This gives us the following specification for a ring:

Ring $R = [\Sigma, \omega]$

Σ: R, Zero ε R

ω: Sum: R x R → R

Opp: R → R

Mult: R x R → R

Axioms:

1. $Sum(r_1, Sum(r_2, r_3)) = Sum(Sum(r_1, r_2), r_3)$
2. $Sum(Zero, r) = r$
3. $Sum(r, Zero) = r$
4. $Sum(r, Opp(r)) = Zero$
5. $Sum(r_1, r_2) = Sum(r_2, r_1)$
6. $Mult(r_1, Mult(r_2, r_3)) = Mult(Mult(r_1, r_2), r_3)$
7. $Mult(r_1, Sum(r_2, r_3)) = Sum(Mult(r_1, r_2), Mult(r_1, r_3))$
8. $Mult(Sum(r_1, r_2), r_3) = Sum(Mult(r_1, r_3), Mult(r_2, r_3))$

If we put Inv: R → R back into our operational specification, add the constant value Unit to R, add the axioms:

9. $Mult(r_1, r_2) = Mult(r_2, r_1)$
10. $Mult(r, Unit) = r$
11. $Inv(Zero) = REJECT$
12. $Mult(r, Inv(r)) = K_{Unit}({}^1r)$ AND $K_{REJECT}({}^2r)$

PARTITION OF r IS

${}^1r | r \neq Zero$ and ${}^2r | r = Zero$

HIGHER ORDER SOFTWARE, INC. · 843 MASSACHUSETTS AVENUE · CAMBRIDGE, MASSACHUSETTS 02139 · (617) 661-8900

to our axiom specification, and remove Axiom 8, because Axiom 9 makes it superfluous, we get a formal specification of the fields as homogeneous algebras.

The REJECT[6] value in Axiom 11 is an object that we assume to be an "invisible" member of every category. Any function that contains REJECT as an argument automatically has REJECT as its value, no matter how deeply embedded the REJECT argument may be. In Axiom 12 we also have the universal primitive operation K_{REJECT}, defined on every type T, that produces the output REJECT for any input and the universal primitive operation K_{Unit} that produces Unit as output for any input (c.f. M. Hamilton and S. Zeldin, "AXES Syntax Description", Higher Order Software, Inc., Cambridge, MA, Dec. 1976, Section 8.0). The REJECT value is useful because it enables us to avoid complicating the operational specifications of particular algebras and the definition of algebra itself. Without the REJECT value, we would have to specify the operation Inv as:

$$\text{Inv: } R - \{\text{Zero}\} \rightarrow R,$$

violating our prescription that the domains and ranges in the operational specifications of the members of ω are always Cartesian products of the members of Σ. We could keep this prescription without the REJECT value only by allowing members of ω to be mathematical partial functions (operations), significantly complicating our notion of algebra.

An algebra which is not homogeneous is said to be <u>heterogeneous</u>. An algebra $[\Sigma,\omega]$ is heterogeneous if Σ contains more than one member. The most familiar heterogeneous algebras are the vector spaces. A non-empty set V is said to be a <u>vector space</u> over a non-empty set S, if (1) V is a commutative group with respect to an addition operation VSum defined on it, (2) S is a field with respect to addition (Sum) and multiplication (Mult) operations defined on it, (3) there is an operation SMult, meaning

[6] Guttag (GUT75) calls this the "error" value, but "error" has overly restrictive connotations. "REJECT" connotes only the fact that a "normal" output is not produced, not that a genuine <u>error</u> has occurred.

scalar multiplication, defined on both V and S which is associative and distributive over both VSum and Sum, and which has the unit element of Mult as its own unit element.

Formally, we get the following specification of the vector spaces as heterogeneous algebras:

Vector Space $v = [\Sigma, \omega]$

Σ: V, VZero $\in$ V

S, S $\neq$ V, Zero $\in$ S, Unit $\in$ S

ω: VSum : V x V $\rightarrow$ V

VOpp : V $\rightarrow$ V

Sum : S x S $\rightarrow$ S

Opp : S $\rightarrow$ S

Mult : S x S $\rightarrow$ S

Inv : S $\rightarrow$ S

SMult: S x V $\rightarrow$ V

Axioms:

1. VSum $(v_1$, VSum$(v_2,v_3))$ = VSum (VSum$(v_1,v_2),v_3)$

2. VSum (VZero,v) = v

3. VSum (v,VZero) = v

4. VSum (v,VOpp(v)) = VZero

5. VSum (v_1,v_2) = VSum (v_2,v_1)

6. Sum $(s_1$,Sum$(s_2,s_3))$ = Sum (Sum$(s_1,s_2),s_3)$

7. Sum (Zero,s) = s

8. Sum (s,Zero) = s

9. Sum (s,Opp(s)) = Zero

10. $\text{Sum}(s_1,s_2) = \text{Sum}(s_2,s_1)$

11. $\text{Mult}(s_1,\text{Sum}(s_2,s_3)) = \text{Sum}(\text{Mult}(s_1,s_2), \text{Mult}(s_1,s_3))$

12. $\text{Mult}(s_1,s_2) = \text{Mult}(s_2,s_1)$

13. $\text{Mult}(s_1,\text{Mult}(s_2,s_3)) = \text{Mult}(\text{Mult}(s_1,s_2),s_3)$

14. $\text{Mult}(\text{Unit},s) = s$

15. $\text{Inv}(\text{Zero}) = \text{REJECT}$

16. $\text{Mult}(s,\text{Inv}(s)) = K_{\text{Unit}}({}^1s)$ AND $K_{\text{REJECT}}({}^2s)$
PARTITION OF s IS
${}^1s \mid s \neq \text{Zero}$
${}^2s \mid s = \text{Zero}$

17. $\text{SMult}(\text{Unit},v) = v$

18. $\text{SMult}(s,\text{VSum}(v_1,v_2)) = \text{VSum}(\text{SMult}(s,v_1), \text{SMult}(s,v_2))$

19. $\text{SMult}(\text{Sum}(s_1,s_2),v) = \text{VSum}(\text{SMult}(s_1,v), \text{SMult}(s_2,v))$

20. $\text{SMult}(\text{Mult}(s_1,s_2),v) = \text{SMult}(s_1, \text{SMult}(s_2,v))$

Axioms 1-5 say that V and its operations comprise an additive Abelian group[7]. Axioms 6-10 say that S, Sum, Zero, and Opp also constitute an additive Abelian group. These axioms plus Axioms 11-14 say that S and its operations comprise a commutative ring and, together with Axioms 15 and 16, that they comprise a field. Axioms 17-20 characterize the operation that relates S and V and, together with the immediately preceding axiomatizations of S and V, say that S,V, and all of the primitive operations specified in the operational specification comprise a vector space.

[7] Or module, but see Note 4.

As specified, the algebra $V = [\Sigma, \omega]$ is a heterogeneous algebra, because Σ contains two distinct categories. If we remove the stipulation that $S \neq V$, then V can be a vector space over itself and the possibility that V could be homogeneous is opened up.

One variety of algebra that has proven to be particularly useful in system specification and that is incorporated into AXES are the type algebras of Guttag (GUT75). If we examine closely the algebras we have seen so far, we realize that what they actually provide are schemata for structured sets. The symbols "G", "R", "V", and "S" in the algebras G, R, and V are set variables, not names of specific sets. Any set can be substituted for these variables, provided that there are operations definable on that set that satisfy the operational and axiomatic specifications. It follows that what these algebras define are mathematical structures, imposable on a large class of otherwise different sets.

In the case of a type algebra we shift our perspective and view the algebra as characterizing a particular set. One of the categories in Σ is singled out as the type-of-interest and the specification of the algebra is interpreted as an implicit definition of the kind of object that makes up the members of the type-of-interest. The categories other than the type-of-interest are also referred to as types in a type algebra.

Guttag (GUT75), for example, gives an algebraic specification of the type *Natural Number* that can be formulated in our framework as follows:

Type *Natural Number* = $[\Sigma, \omega]$

Σ: Natural Number, Zero ε Natural Number
Boolean, True ε Boolean, False ε Boolean

ω: Succ: Natural Number → Natural Number
?Zero?: Natural Number → Boolean
?Equal?: Natural Number x Natural Number → Boolean
?>?: Natural Number x Natural Number → Boolean

Axioms:

1. ?Zero? (Zero) = True
2. ?Zero? (Succ(n)) = False
3. ?Equal? (Zero, Zero) = True
4. ?Equal? (Succ(n),Zero) = False
5. ?Equal? (Zero, Succ(n)) = False
6. ?Equal? (Succ(n)), Succ(n_1)) = ?Equal? (n,n_1)
7. ?>? (Zero,Zero) = False
8. ?>? (Succ(n),Zero) = True
9. ?>? (Zero, Succ(n)) = False
10. ?>? (Succ(n), Succ(n_1)) = ?>? (n,n_1)

Guttag introduces Zero as an operation that maps the empty set onto Natural Number, Zero: $\emptyset \rightarrow$ Natural Number, saying that the emptiness of $\emptyset$ guarantees that a unique constant value results. Actually, however, since a mapping is mathematically a set of ordered pairs the first element of each member of which is a member of the domain, it follows that having an empty domain guarantees that there is no first element of any ordered pair and thus no ordered pairs. The mapping, viewed as a set of ordered pairs, turns out to be the empty set and thus not really a mapping at all. If there is <u>no</u> input, there can be <u>no</u> output, unique, constant, or otherwise. Guttag's device is really unnecessary, in any event, because all we have to do is to state in our specification of I that Zero is in Natural Number. With AXES, this is done by means of a WHERE statement, as we will see in Appendix IV.

The intent of the operations in this specification should be clear from their names, but their formal meaning is provided entirely by their operational and axiomatic specifications. Guttag's specification of the type Natural Number is, in reality, inadequate, because it omits the crucial axiom of induction; it still serves our purpose, however, as an example. We will see in Appendix IV how the inadequacy of his formulation can be remedied.

In this example, we see that a type algebra can be viewed as defining what it means to be a member of the set with the same name. The algebra *Natural Number* defines the set of natural numbers. An object is a natural number if it belongs to the set characterized by the respective algebra. The primitive operations in such an algebra are taken as being defined collectively in terms of their interactive behavior. The function Succ, meaning successor, for example, has meaning only with respect to the specification of *Natural Number* as a whole. The specification defines all of its primitive operations at the same time, each in terms of the others, and, through them it defines its type-of-interest. What makes *Natural Number* a type algebra, in contrast to the general algebras we saw earlier, is that the type-of-interest is taken to be a specific set of a particular kind of object that already exists in the world (or in our minds) and which we are trying to make intelligible. The general algebras we saw earlier define mathematical structures on arbitrary sets, as we have seen. Guttag (GUT75) characterizes the type algebra as a restricted form of the general heterogeneous algebra and implies that the type Boolean must be presupposed. Our explicit specification in Appendix IV of *Boolean* as a homogeneous type algebra, however, shows that both claims are incorrect.

The usefulness of type algebras for system specification lies in the need to maximize the degree of abstraction in the specification of data types. The customary operational means of specifying abstract data types requires us to imbue our data types

with more implementational meaning than is often desirable. If we have to include elements of implementation in specifying an abstract data type, then we may unwittingly rule out more efficient implementations of that data type that are inconsistent with those elements. With HOS, however, we are able to divorce specification entirely from implementation and, with respect to abstract data types, we manage to do this by specifying those types algebraically, rather than operationally. Appendix IV contains a list of algebraic specifications of abstract data types that are included in AXES as intrinsic types.

APPENDIX III BIBLIOGRAPHY

BIR70 Birkhoff, G., and Lipson, J.D., "Heterogeneous Algebras," Journal of Combinatorial Theory 8, 1970

FUN74 Fundamentals of Mathematics, Vol I: Foundations of Mathematics/The Real Number System and Algebra, ed. H. Behnke, et. al., trans. S.H. Gould, The MIT Press, Cambridge, MA, 1974.

GUT75 Guttag, J., "The Specification and Application to Programming of Abstract Data Types," Univ. of Toronto Technical Report CSRG-59, September 1975.

HIGHER ORDER SOFTWARE, INC. · 843 MASSACHUSETTS AVENUE · CAMBRIDGE, MASSACHUSETTS 02139 · (617) 661-8900

APPENDIX IV

THE INTRINSIC TYPES OF AXES

by S. Cushing

APPENDIX IV
THE INTRINSIC TYPES OF AXES

Although AXES provides the means for algebraically specifying any desired abstract data type, there are a few types that are of sufficiently general usefulness in a wide variety of systems that we include them in AXES as intrinsic types. These types must also be specified algebraically, of course, and we do so, once and for all, in this Appendix. In all we provide six intrinsic data types in AXES: Booleans, properties, sets, natural numbers, integers, and rational numbers. The Boolean data type automatically solves the "boot-strap" problem for abstract data types, because it can be characterized as a homogeneous algebra, i.e., entirely in terms of itself. All our other intrinsic types presuppose the prior characterization of type *Boolean* and so must be characterized as heterogeneous algebras. Type *Property* presupposes only type *Boolean*, while type *Set* and type *Natural* all presuppose type *Property*. Type *Integer* presupposes type *Natural*, and type *Rational* presupposes type *Integer*. Our reasons for not providing the real numbers as an intrinsic data type will be discussed in connection with our algebraic specification of the rationals. All our intrinsic types will be specified in AXES syntax, rather than in the strictly mathematical format used in Appendix III. In this Appendix, AXES statements are often numbered for purposes of discussion (these numbers are not intended to be included as part of the AXES syntax).

Type *Boolean* is particularly easy to characterize, because it contains only two values, true and false (truth and falsity). Since Σ contains only one set and that set is finite, we could identify that set explicitly by simply listing its members. This frees us from the need to characterize the equality relation on type *Boolean*, which we could not do without a prior characterization of type *Property*. Since there are only two distinct Booleans, which we explicitly list in the category specification, we can always tell which one we are dealing with simply by looking at it:

```
DATA TYPE:  BOOLEAN;
PRIMITIVE OPERATIONS:

boolean3 = And(boolean1,boolean2);                    1.

boolean2 = Not(boolean1);                             2.

AXIOMS:
     WHERE True IS A CONSTANT BOOLEAN;
     WHERE False IS A CONSTANT BOOLEAN;

And(True,True) = True;                                1.

And(True,False) = False;                              2.

And(False,True) = False;                              3.

And(False,False) = False;                             4.

Not(True) = False;                                    5.

Not(False) = True;                                    6.

END BOOLEAN;
```

In this algebra we specify that Σ's single member contains exactly two elements, true and false, and that ω contains exactly two primitive operations, And and Not. And is characterized as behaving exactly like the conjunction operator of propositional logic and Not is characterized as behaving like the negation operator. These two elements and these two operations, as axiomatized, are all we need to characterize the type *Boolean* as an abstract data type.

Once we have characterized an abstract data type in terms of its categories and its primitive operations, defined collectively and implicitly through its axioms, we will often find it useful to define other operations on that type. Note that the categories of the type algebra *Boolean* other than Boolean itself were not listed explicitly in the AXES specification above, but only implicitly through their appearance in the PRIMITIVE OPERATION specification.

We are free to define any operation we want on an already-defined type as long as the operation definition is consistent with the axioms of the type. New operations can be characterized either as operations (c.f. HAM76, Section 8) or as derived operations (c.f. HAM76, Section 16). An operation is specified in AXES explicitly in a form that is directly translatable to a control map. A derived operation is specified implicitly by means of assertions that describe the behavior of the operation with respect to other already-defined operations. Either kind of operation could be written as a control map, if desired. They differ in how they are specified, not in what they are. What distinguishes both of these kinds of operations from primitive operations on their data type is that their existence is provable mathematically from the existence of the primitive operations and the axioms of the type. If an operation's existence is not so provable, then adding it to the type produces a new type, of which the new operation is a primitive.

In the case of type *Boolean*, for example, we will often find it useful, as in logic, to have available the notions of disjunction, entailment, and sameness of truth-value. We can introduce these notions as operations on type *Boolean* by means of the following definitions:

```
OPERATION:  b3 = Or(b1,b2);
WHERE b1,b2,b3 ARE BOOLEANS;
WHEREBY b3 = Not(And(Not(b1),Not(b2)));
END Or;

OPERATION:  b3 = Entails(b1,b2);
WHERE b1,b2,b3 ARE BOOLEANS;
WHEREBY b3 = Or(Not(b1),b2);
END Entails;

OPERATION:  b3 = Same(b1,b2)
WHERE b1,b2,b3 ARE BOOLEANS;
WHEREBY b3 = Or(And(b1,b3),And(Not(b1),Not(b2)));
END Same;
```

The first definition defines Or in terms of And and Not in a way that is familiar from propositional logic. We could have introduced it as a primitive operation by including the axioms:

```
Or (True,True) = True;
Or (True,False) = True;
Or (False,True) = True;
Or (False,False) = False;
```

in our axiomatic specification of type *Boolean*, but this would have complicated our algebra unnecessarily. We simply do not need Or to characterize the Booleans as a data type. Similarly, we could have included Entails and Same as primitive operations, but there was no point in doing so as long as we can define them as operations. The point is that And and Not are all we need to characterize the Booleans, even though there are other operations that we find useful, and that we therefore introduce for other purposes.

It should be noted that Same is an equivalence relation on type *Boolean*. This relation coincides with equality, because we already know when two Booleans are the same or distinct, as a result, as noted above, of the finiteness of the single set in Σ. If this were not the case, in fact, that is, if equality were not automatically given to us, then it would be impossible to write axioms for type *Boolean*, because the "=" sign would be meaningless. For convenience and clarity, we will sometimes use "=", and a few other symbols like "<", in the conventional way, rather than in strictly functional notation, once we have already defined them functionally. For example, it may be simpler to write "$i_1<i_2$" in an axiom for type *Rational*, rather than "$I<(i_1,i_2)$." This is permissable, because we will already have characterized I< (i.e., "integer-less than") in our specification of type *Integer* (c.f. HAM76 (Section 11)).

The second intrinsic data type that AXES provides is that of properties. We will need properties in characterizing the sets as a data type. Properties are basically things that map other things onto truth-values, i.e., Booleans. The property "prime", for example, maps the integer 2 onto true, 3 onto true, and 4 onto false, because 2 is prime and 3 is prime, but 4 is not prime. In characterizing properties algebraically, we will have to state what kinds of things the properties are properties of. We can do this by including a type parameter "T" in our category specification and treating our algebraic specification as a function of *T*. It follows that our algebra for type *Property* is really an algebra schema depending of the type parameter *T* and that there is, therefore, a distinct type *Property (of T)* for every type *T*.

We can express the fact that properties map other things (i.e., t's) onto Booleans by introducing a function that maps properties and t's onto Booleans. If we call this function "Has", so that "Has(P,t)" is true, when t has the property P, then we must specify that Has maps properties and t's onto Booleans in a way that preserves conjunctions and negations. This can be stated very simply in terms of axioms. To define equality or identity of properties, we will also have to introduce two quantifier operations Forall and Exists (CUS76a). Properties can be mapped onto Booleans by combining them with t's via the Has function, but they can also be mapped onto Booleans directly via these quantifier functions. Has maps P and t onto true, if t has the property P. Forall maps P itself onto true, if every t has the property P. Exists, similarly, maps P onto true, if there is some t that has P, regardless of which particular t that is. Once we have Forall available to us, it will be a simple matter to specify when two properties are equal (identical).

As well as characterizing the relationship, which we have just discussed, between type *Property (of T)* and type *Boolean*, we must also characterize the internal structure of type *Property (of T)*. Properties constitute a Boolean lattice (FUN74), so we

must include the axioms for a Boolean lattice in their algebraic specification as a data type. The Booleans also constitute a Boolean lattice, but since there are only two Booleans, enabling us to list the values of their primitive operations explicitly, we can prove the axioms for a Boolean lattice from that explicit list of values. For properties, however, we must include the axioms for a Boolean lattice as axioms of our algebra, because there is nothing else that we can prove them from.

The foregoing discussion is summarized (and elaborated) in the following AXES specification:

DATA TYPE: PROPERTY(OF T);
PRIMITIVE OPERATIONS:

$property_3$ = Pand($property_1$,$property_2$); 1.

$property_3$ = Por($property_1$,$property_2$); 2.

$property_2$ = Pnot($property_1$); 3.

$property_3$ = Pentails($property_1$,$property_2$); 4.

boolean = Has(property,t); 5.

boolean = Forall(property); 6.

boolean = Exists(property); 7.

boolean = Ident($property_1$,$property_2$); 8.

AXIOMS:

WHERE T IS SOME TYPE;
WHERE P_1,P_2,P_3 ARE PROPERTIES;
WHERE t is a T;
WHERE Nec IS A CONSTANT PROPERTY;
WHERE Contra IS A CONSTANT PROPERTY;

Pand(P_1,P_2) = Pand(P_2,P_1); 1.

Por(P_1,P_2) = Por(P_2,P_1); 2.

Pand(P_1,Pand(P_2,P_3)) = Pand(Pand(P_1,P_2),P_3); 3.

Por(P_1,Por(P_2,P_3)) = Por(Por(P_1,P_2),P_3); 4.

```
Pand(P1,Por(P1,P2)) = P1;                                          5.
Por(P1,Pand(P1,P2)) = P1;                                          6.
Pand(P1,Por(P2,P3)) = Por(Pand(P1,P2), Pand(P1,P3));               7.
Por(P1,Pand(P2,P3)) = Pand(Por(P1,P2), Por(P1,P3));                8.
Pand(P,Pnot(P)) = Contra;                                          9.
Por(P,Pnot(P)) = Nec;                                             10.
Has(Nec,t) = True;                                                11.
Has(Contra,t) = False;                                            12.
Has(Pand(P1,P2),t) = And(Has(P1,t), Has(P2,t));                   13.
Has(Por(P1,P2),t) = Or(Has(P1,t), Has(P2,t));                     14.
Has(Pnot(P),t) = Not(Has(P,t));                                   15.
Forall(Nec) = True;                                               16.
Exists(Contra) = False;                                           17.
Forall(P) = Not(Exists(Pnot(P)));                                 18.
Exists(P) = Not(Forall(Pnot(P)));                                 19.
Entails(Forall(P), Same(Has(P,t),True)) = True;                   20.
Entails(Same(Has(P,t),True), Exists(P)) = True;                   21.
Ident(P1,P2) = And(Forall(Pentails(P1,P2)),
                   Forall(Pentails(P2,P1)));                      22.
END PROPERTY(OF T)

OPERATION:  P3 = Pentails(P1,P2);
WHERE P1,P2,P3 ARE PROPERTIES;
WHEREBY P3 = Por(Pnot(P1),P2);
END Pentails;
```

Axioms 1-10 in this specification characterize type *Property (of T)* as a Boolean lattice, and together with Axiom 22, give us the internal structure of the type. Axiom 22 is essential to the internal structure, because it tells us when two properties

are the same and when they are distinct. The value Nec is the necessary property, which every t has, and serves as the unit element of the lattice, while the value Contra is the contradictory property, which no t has, and serves as the zero element of the lattice. Axioms 18 and 19 tell us that Forall and Exists are related by dual negation, which is definable for any quantifier (CUS76a, CUS76b). Axioms 11-21 characterize the interface of type *Property(of T)* with type *Boolean*, but they also provide the prerequisite for the meaningfulness of Axiom 22. We thus see the sort of mutual dependence among the various aspects of specification, in this case between the internal structure and the external interface, that is characteristic of algebraic specification. One might think that Ident could be defined as an operation, since Axiom 22 defines it explicitly in terms of already defined operations. This would be wrong however, because a notion of identity (equality) is essential to characterizing the internal structure of the type. Without Axiom 22, Axioms 1-10 would literally be meaningless, because we would have no clearly specified interpretation of the "=" signs that occur in them.

We have stated (in Axiom 22) that two properties are identical if they are mutually entailing for every member of the type whose members they are properties of, that is, if they hold of exactly the same members of that type. Ultimately, such a definition is inadequate, because it treats certain properties as identical which, for some purposes, should not be considered identical. The two conjunctive properties, "both less than and greater than 2" and "both less than and greater than 100", for example, are distinct properties, in the general sense, because they "say different things" about the objects they are supposed to hold of. By our definition, however, these two properties are identical and, in fact, are both identical to Contra, because they hold of exactly the same objects, namely none. Since we are interested in properties primarily as a way of specifying set partitions in system specifications, our definition of identity nevertheless suffices for our purposes.

Given the way we have characterized properties as an abstract data type, it is a simple matter to do the same for sets. Because of the way we have defined identity for properties, the type *Property (of T)*, as we have specified it, will be isomorphic to the type *Set (of T)*. For every property there is a set, called the extension of that property, which consists of exactly the objects that have that property. Given our definition of property identity, this mapping from properties to sets is one-to-one. It follows that we can characterize type *Set (of T)* isomorphically to type *Property (of T)* in terms of this extension mapping, if we guarantee that the mapping and its inverse preserve the primitive operations of the two types. This is done in the following specification:

DATA TYPE: SET(OF T);
PRIMITIVE OPERATIONS:

set_3 = Inters(set_1,set_2); 1.

set_3 = Union(set_1,set_2); 2.

set_2 = Comp(set_1); 3.

set = Extension(property); 4.

property = Prop(set); 5.

boolean = Element(t,set); 6.

boolean = Subset(set_1,set_2); 7.

boolean = Equal(set_1,set_2); 8.

AXIOMS:

WHERE s_1,s_2,s_3 ARE SETS;
WHERE P IS A PROPERTY;
WHERE Null IS A CONSTANT SET;
WHERE T IS SOME TYPE;

Inters(s_1,s_2) = Inters(s_2,s_1); 1.

Union(s_1,s_2) = Union(s_2,s_1); 2.

Inters(s_1,Inters(s_2,s_3))= Inters(Inters(s_1,s_2),s_3); 3.

Union(s_1,Union(s_2,s_3)) = Union(Union(s_1,s_2),s_3); 4.

```
Inters(s1,Union(s1,s2)) = s1;                                    5.
Union(s1,Inters(s1,s2)) = s1;                                    6.
Inters(s1,Union(s2,s3)) = Union(Inters(s1,s2),
                                Inters(s1,s3));                  7.
Union(s1,Inters(s2,s3)) = Inters(Union(s1,s2),
                                 Union(s1,s3));                  8.
Inters(s,Comp(s)) = Null;                                        9.
Union(s,Comp(s)) = T;                                           10.
Extension(Prop(s)) = s;                                         11.
Prop(Extension(P)) = P;                                         12.
Prop(T) = Nec;                                                  13.
Prop(Null) = Contra;                                            14.
Prop(Inters(s1,s2)) = Pand(Prop(s1),Prop(s2));                  15.
Prop(Union(s1,s2)) = Por(Prop(s1), Prop(s2));                   16.
Prop(Comp(s)) = Pnot(Prop(s));                                  17.
Element(t,s) = Has(Prop(s),t);                                  18.
Subset(s1,s2) = Forall(Pentails(Prop(s1),Prop(s2)));            19.
Equal(s1,s2) = And(Subset(s1,s2), Subset(s2,s1));               20.
END SET(OF T);
```

Axioms 1-10 in this specification characterize type *Set of (T)* as a Boolean lattice, with the null set Null as the zero element and the universal set T as the unit element. Axioms 11-17 define the isomorphism mapping between type *Set (of T)* and type *Property (of T)*. The function Extension maps a property onto the set of elements that have that property, and the function Prop, meaning "property", maps a set onto the property of being in that set. This automatically accounts for all properties because of our definition of property identity, as noted above. Axioms 18 and 19 define the usual notions of element and subset, and Axiom 20 defines equality as mutual subset. Something is in a set if it

has the property that corresponds to the set and one set is a subset of a second if everything that has the property of the first has the property of the second. Two sets are equal if each is a subset of the other. It is worth noting that T itself is a grandmember of Σ in this case, because it functions as the unit element of the algebra. Upon reflection, we realize that the set Set (of T), i.e., the member of Σ, as opposed to the algebra, turns out to be just the power set of T itself.

Now that we have sets and properties available to us, we can construct an adequate specification of the natural numbers as an abstract data type. As we noted in Appendix III, Guttag's specification of the type *Natural Number* is inadequate, because it leaves out the crucial axiom of induction. This axiom can be formulated as follows (FUN74, p. 72):

> If a property P of the natural numbers satisfies the following two conditions, then P holds for every natural number:
>
> (1) P holds for 0
>
> (2) For every natural number n, if P holds for n, then P holds for n';

where n' is the successor of n. This axioms tells us that we can be sure every natural number has a given property, if we know that 0 has that property and that n+1's having it follows from n's having it, for every n. If we begin at 0, in other words, and go successively from each natural number to the next, then we eventually get to every natural number. This is a crucial characteristic of the natural numbers and cannot be omitted if our intent is to characterize their data type as fully as possible.

Since we now have the facility for dealing with properties, we could formalize the axiom of induction as an axiom of type *Natural Number* in terms of the members of type *Property (of Natural Number)*, by taking T = *Natural Number*, in other words, in our type *Property (of T)*. It turns out, however, that the actual formulation of this axiom in our framework is very complicated and somewhat unintuitive, so we are led to look for an alternative axiom that would

have the same effect as the axiom of induction. Fortunately, this purpose can be served by a characteristic of the natural numbers called the "well-ordering principle", which states that every non-empty set of natural numbers contains a least element. The axiom of induction and the well-ordering principle are logically equivalent, in the sense that each can be derived from the other within the context of the other axioms for the natural numbers (LAN67), so we are free to take either one as one of our axioms. The well-ordering principle can be formulated very simply in our framework, in contrast to the complexity and unintuitive character of the axiom of induction, so we will adopt it to complete our specification of type *Natural Number*.

This gives us the following AXES specification:

DATA TYPE: NATURAL;
PRIMITIVE OPERATIONS:

natural$_2$ = Succ(natural$_1$); 1.

boolean = ?Zero?(natural); 2.

boolean = ?Equal?(natural$_1$,natural$_2$); 3.

boolean = ?>?(natural$_1$,natural$_2$); 4.

natural = Smin(set(of naturals)$_1$); 5.

AXIOMS:

WHERE n,n$_1$ ARE NATURALS;
WHERE s IS A SET(OF NATURALS);
WHERE Zero IS A CONSTANT NATURAL;

?Zero?(Zero) = True; 1.

?Zero?(Succ(n)) = False; 2.

?Equal?(Zero,Zero) = True; 3.

?Equal?(Succ(n),Zero) = False; 4.

?Equal?(Zero,Succ(n)) = False; 5.

?Equal?(Succ(n),Succ(n$_1$)) = ?Equal?(n,n$_1$); 6.

?>?(Zero,Zero) = False; 7.

```
?>?(Succ(n),Zero) = True;                                    8.

?>?(Zero,Succ(n)) = False;                                   9.

?>?(Succ(n),Succ(n1)) = ?>(n,n1);                           10.

Element(Smin(s),s) = True;                                  11.

Entails(Element(n,s), ?>?(n,Smin(s))) = True;               12.

END NATURAL;
```

This specification is identical to Guttag's specification of type *Natural Number*, which we saw in Appendix III, except for the new operation Smin and the two new Axioms 11 and 12. Axioms 11 and 12, along with the presence of Smin in ω, provide us with the effect of the well-ordering principle. The fact that the Smin is in ω tells us that every set s of natural numbers is associated with a natural number Smin(s). Axiom 11 tells us that the natural Smin(s) is an element of s and Axiom 12 tells us that Smin(s) is, in fact, the minimum element of s. This specification, then, completely specifies type *Natural Number* as the type of what we usually think of as the natural numbers.

Now that we have a full specification of the natural numbers, we can define operations on their data type. Since we have already characterized equality of natural numbers as a primitive operation of our data type, we are free to interpret the "=" sign in our definitions as referring to that equality. We will also use other operations, such as "And" in the customary way, rather than the more complicated functional notations, as long as these operations have been fully characterized (cf. section 10). Some of the following operations, such as Sum and Prod, meaning sum and product, respectively, are included because of their general usefulness; others are included because they will be useful in specifying later data types:

```
DERIVED OPERATION:  n3 = Sum(n1,n2);
     WHERE n1,n2,n3 ARE NATURALS;

Sum(Zero,n2) = n2;                                           1.

Sum(n1,Zero) = n1;                                           2.
```

```
Sum(n1,Succ(n2)) = Succ(Sum(n1,n2));                    3.
Sum(Succ(n1),n2) = Succ(Sum(n1,n2));                    4.
END Sum;

DERIVED OPERATION:  n3 = Prod(n1,n2);
WHERE n1,n2,n3 ARE NATURALS;
Prod(Zero,n2) = Zero;                                   1.
Prod(n1,Zero) = Zero;                                   2.
Prod(n1,Succ(n2)) = Sum(Prod(n1,n2),n1);                3.
Prod(Succ(n1),n2) = Sum(Prod(n1,n2),n2);                4.
END Prod;

DERIVED OPERATION:  n3 = Ndiff(n1,n2);
WHERE n1,n2,n3 ARE NATURALS;
Sum(n1,Ndiff(1(n1,n2)) = n2;                            1.
Ndiff(2(n1,n2)) = REJECT;                               2.
PARTITION OF (n1,n2) IS
     1(n1,n2) | n1≥n2,
     2(n1,n2) | n2>n1;
END Ndiff;

DERIVED OPERATION:  n3 = Max(n1,n2);
WHERE n1,n2,n3 ARE NATURALS;
Max 1(n1,n2) = n1;                                      1.
Max 2(n1,n2) = n2;                                      2.
PARTITION OF (n1,n2) IS
     1(n1,n2) | n2≤n1,
     2(n1,n2) | n1<n2;
END Max;
```

DERIVED OPERATION: n_3 = Min(n_1,n_2);
WHERE n_1,n_2,n_3 ARE NATURALS;

Min($^1(n_1,n_2)$) = $^1n_2$; 1.

Min($^2(n_1,n_2)$) = $^2n_1$; 2.

PARTITION OF (n_1,n_2) IS

$^1(n_1,n_2)|n_2 \leq n_1$,

$^2(n_1,n_2)|n_1 < n_2$;

END Min;

DERIVED OPERATION: n_3 = Quot(n_1,n_2);
WHERE n_1,n_2,n_3 ARE NATURALS;

Quot $^1(n_1,n_2)$ = REJECT; 1.

Sum(Prod(Quot $^2(n_1,n_2)$, Rem $^2(n_1,n_2)$))= n_1; 2.

PARTITION OF (n_1,n_2) IS

$^1(n_1,n_2)|n_2 = 0$,

$^2(n_1,n_2)|n_2 \neq 0$;

END Quot;

DERIVED OPERATION: n_3 = GCD(n_1,n_2);
WHERE n_1,n_2,n_3 ARE NATURALS;

Factor(GCD(n_1,n_2),n_1) = True; 1.

Factor(GCD(n_1,n_2),n_2) = True; 2.

Entails(And(And(Factor(n_1,n_2),Factor(n_1,n_3)),Not(?Equal?(n_1,Zero))),
Factor(n_1,GCD(n_2,n_3))) = True; 3.

END GCD;

OPERATION: n_3 = Rem(n_1,n_2);
WHERE n_1,n_2,n_3 ARE NATURALS;

EITHER n_3 = $K_{REJECT}(^1n_1, ^1n_2)$ OTHERWISE

EITHER n_3 = $IDENTIFY_1^2(^2n_1, ^2n_2)$ OTHERWISE

WHEREBY n_3 = Rem(Ndiff $^3(n_1,n_2)$, $^3n_2$;

PARTITION OF (n_1,n_2) IS

$^1(n_1,n_2) | n_2 = 0$,

$^2(n_1,n_2) | n_2 \neq 0$ AND $n_1 < n_2$,

$^3(n_1,n_2) | n_2 \neq 0$ AND $n_2 \leq n_1$;

END Rem;

OPERATION: b = Factor(n_1,n_2);
WHERE n_1,n_2 ARE NATURALS;
WHERE b IS A BOOLEAN;
WHEREBY b = ?Equal?(Rem(n_2,n_1),Zero);
END Factor;

Derived operations Sum and Prod give us addition and multiplication, respectively. Derived operation Ndiff gives us the subtraction of smaller naturals from larger ones. Derived operation Max gives us the larger of two naturals, derived operation Rem gives us division (quotient) with remainder, and operation Factor tells us when one natural is a factor of another. Derived operation GCD gives us the greatest common divisor of two naturals and will be needed in the specification of the rationals.

The Integers can be characterized as a data type in terms of the natural numbers by recognizing that an integer is just a natural number with a sign. Since we need two distinct signs, we can take our signs to be the Booleans, with True interpreted as plus and False interpreted as minus. This gives us the following specification:

DATA TYPE: INTEGER;
PRIMITIVE OPERATIONS:

boolean = ?Iequal?(integer$_1$,integer$_2$); 1.
boolean = ?I>?(integer$_1$,integer$_2$); 2.
natural = Abs(integer); 3.

boolean = Sign(integer); 4.

$integer_3$ = Isum($integer_1$,$integer_2$); 5.

$integer_3$ = Iprod($integer_1$,$integer_2$); 6.

$integer_3$ = Iquot($integer_1$,$integer_2$); 7.

AXIOMS:

WHERE i_1,i_2 ARE NATURALS;

WHERE Izero IS A CONSTANT INTEGER;

WHERE Ione IS A CONSTANT INTEGER;

?Iequal?(i_1,i_2) = Or(And(?Equal?(Abs(i_1),Zero),
?Equal?(Abs(i_2),Zero)),
And(?Equal(Abs(i_1),Abs(i_2)),
Same(Sign(i_1),Sign(i_2)))); 1.

?I>?(i_1,i_2) = (Same(Sign(i_1),True) & Same(Sign(i_2),True)
&?>?(Abs(i_1),Abs(i_2)))
!(Same(Sign(i_1),False) & Same(Sign(i_2),False)
&?>?(Abs(i_2),Abs(i_1)))
!(Same(Sign(i_1),True) & Same(Sign(i_2),False)); 2.

Abs(Isum(i_1,i_2)) = Sum(Abs($^1i_1$),Abs($^1i_2$)) AND
(Ndiff(Max(Abs($^2i_1$), Abs($^2i_2$)),
Min(Abs($^2i_1$),Abs($^2i_2$))); 3.

PARTITION OF (i_1,i_2) IS

^{1}(i_1,i_2)|Sign(i_1) = Sign(i_2),

^{2}(i_1,i_2)|Sign(i_1) ≠ Sign(i_1);

```
Sign(Isum(i1,i2)) = Sign(1i1) AND Sign(2i1) AND Sign(3i2);    4.

PARTITION OF (i1,i2) IS

    1(i1,i2)|Sign(i1) = Sign(i2),

    2(i1,i2)|Sign(i1) ≠ Sign(i2) AND Abs(i2) ≤ Abs(i1),

    3(i1,i2)|Sign(i1) ≠ Sign(i2) AND Abs(i1) < Abs(i2);

Abs(Iprod(i1,i2)) = Prod(Abs(i1),Abs(i2));                    5.
Sign(Iprod(i1,i2)) = Same(Sign(i1),Sign(i2));                 6.
Abs(Izero) = Zero;                                            7.
Sign(Izero) = True;                                           8.
Abs(Ione) = Succ(Zero);                                       9.
Sign(Ione) = True;                                           10.
Abs(Iquot(i1,i2)) = Quot(Abs(i1),Abs(i2));                   11.
Sign(Iquot(i1,i2)) = Same(Sign(i1),Sign(i2));                12.
END INTEGER;

DERIVED OPERATION:  integer2 = Iopp(integer1);
WHERE i IS AN INTEGER;
Sum(i,Iopp(i)) = Izero;
END Iopp;

OPERATION:  i3 = Idiff(i1,i2);
WHERE i1,i2,i3 ARE INTEGERS;
WHEREBY i3 = Sum(i1,Iopp(i2));
END Idiff;

DERIVED OPERATION:  integer3 = IGCD(integer1,integer2);
WHERE i1,i2 ARE INTEGERS;
Abs(IGCD(i1,i2)) = GCD(Abs(i1),Abs(i2));                      1.
Sign(IGCD(i1,i2)) = True;                                     2.
END IGCD;
```

Axiom 1 is complicated by the fact that zero can have either a plus (true) or minus (false) sign. We want Sign to be a mapping, however, (i.e., a function in the mathematical, not AXES, sense, c.f. HAM76 (Section 8.0)) so we assume from the start that plus zero and minus zero are the same entity. In Axiom 8 we say zero has a plus sign, but Axiom 1 tells us that if a minus zero occurs, it is really the same integer as plus zero. Two integers are equal if they have the same absolute value and the same sign, unless their absolute values are both zero. In that case, they are equal regardless of their signs.

The rational numbers can be characterized, as in modern arithmetic theory, as ordered pairs of integers that have no common factors. Adopting this approach we get the following specification:

```
DATA TYPE:  RATIONAL;
PRIMITIVE OPERATION:
boolean = ?Requal?(rational1,rational2);                    1.
boolean = ?R>?(rational1,rational2);                        2.
integer = Num(rational);                                    3.
integer = Denom(rational);                                  4.
rational = Rsum(rational1,rational2);                       5.
rational = Rprod(rational1,rational2);                      6.
boolean = Pos(rational);                                    7.
AXIOMS:
     WHERE r,r1,r2 ARE RATIONALS;
     WHERE Rzero IS A CONSTANT RATIONAL;
     WHERE Rone IS A CONSTANT RATIONAL;
?Iequal?(Denom(r),Izero) = False;                           1.
IGCD(Abs(Num(r),Abs(Denom(r)))) = Ione;                     2.
Rprod(r,Denom(r)) = Num(r);                                 3.
?Requal?(r1,r2) = ?Iequal?(Iprod(Num(r1),Denom(r2)),
                  Iprod(Denom(r1),Num(r2)));                4.
```

```
Num(Rsum(r1,r2)) = Iquot(Cross(r1,r2), IGCD(Abs(Cross(r1,r2)),
                   Abs(Dprod(r1,r2))));                                  5.

Denom(Rsum(r1,r2)) = Iquot(Dprod(r1,r2), IGCD(Abs(Cross(r1,r2)),
                     Abs(Dprod(r1,r2))));                                6.

Num(Rprod(r1,r2)) = Iquot(Nprod(r1,r2), IGCD(Abs(Nprod(r1,r2)),
                    Abs(Dprod(r1,r2))));                                 7.

Denom(Rprod(r1,r2)) = Iquot(Dprod(r1,r2), IGCD(Abs(Nprod(r1,r2)),
                      Abs(Dprod(r1,r2))));                               8.

Num(Rzero) = Izero;                                                      9.
Denom(Rzero) = Ione;                                                     10.

Pos(r) = And(Not(Equal(r,Rzero)),Same(Sign(Num(r)),
         Sign(Denom(r))));                                               11.

?R>?(r1,r2) = Pos(Rdiff(r1,r2));                                         12.

END RATIONAL;

OPERATION:  r3 = Cross(r1,r2);
WHERE r1,r2,r3 ARE INTEGERS;
WHEREBY r3 = Isum(Iprod(Num(r1),Denom(r2)),
             Iprod(Denom(r1),Num(r2)));
END Cross;

OPERATION:  r3 = Nprod(r1,r2);
WHERE r1,r2,r3 ARE INTEGERS;
WHEREBY r3 = Iprod(Num(r1),Num(r2));
END Nprod;

OPERATION:  r3 = Dprod(r1,r2);
WHERE r1,r2,r3 ARE INTEGERS;
WHEREBY r3 = Iprod(Denom(r1),Denom(r2));
END Dprod;
```

The functions Num and Denom give the numerator and denominator of a "fraction" in the "lowest terms". The operations Iquot and GCD are used throughout the axioms to guarantee that sums and products of rationals are always expressed in such "lowest terms". The operations Cross, Nprod, and Dprod are just useful abbreviations that greatly simplify the definitions of addition and multiplication.

```
DERIVED OPERATION:  rational2 = Ropp(rational1);
WHERE r IS A RATIONAL;
Rsum(r,Ropp(r)) = Rzero;
END Ropp;

OPERATION:  r3 = Rdiff(r1,r2);
WHERE r1,r2,r3 ARE RATIONALS;
WHEREBY r3 = Rsum(r1,Ropp(r2));
END Rdiff;

DERIVED OPERATION:  rational2 = Rinv(rational1);
WHERE r,r1,r2 ARE RATIONALS;
Rinv(Rzero) = REJECT;
EITHER Num(Rprod(r1),Rinv(r2))  = K_REJECT(1r)
OTHERWISE Num(Rprod(r1),Rinv(r2)) = K_Ione(2r);
EITHER Denom(Rprod(r,Rinv(r2))) = K_REJECT(1r)
OTHERWISE Denom(Rprod(r,Rinv(r2))) = K_Ione(2r);

PARTITION OF r IS
     1r|r = 0,
     2r|r ≠ 0;
END Rinv;

OPERATION:  r3 = Rdiv(r1,r2);
WHERE r1,r2,r3 ARE RATIONALS;
WHEREBY r3 = Rprod(r1,Rinv(r2));
END Rdiv;
```

These are the usual opposite, difference, inverse, and division for the rational numbers.

The problem of specifying the real numbers presents a serious problem for the algebraic specification techniques introduced by Guttag and expanded here. We have already seen how Guttag's approach must be expanded to give an adequate specification of the natural numbers. A complete account of the natural numbers requires an axiom equivalent to the axiom of induction and well-ordering principle and such an axiom cannot be formulated without a specification of properties or sets as abstract data types, or some equivalent modification of Guttag's approach. In the case of the reals we encounter a similar situation. The principal reason for introducing the real numbers in mathematics is to fill in the "holes," so to speak, in the set of rationals visualized as a "line." Speaking somewhat more formally, there exist sequences of rationals that seem for all the world as if they "ought" to converge, but for which there is no rational to which they do converge. The reals are introduced to provide limits for these otherwise non-convergent sequences. Speaking still more formally, we introduce the following definitions, where K is the set of rationals (actually, any ordered field) (LAN67, pp. 123-4):

> A sequence $\{x_n\}$ in K is said to be a Cauchy sequence if given an element $\in > 0$ in K, there exists a positive integer N such that for all integers m, $n \geq N$ we have
>
> $$|x_n - x_m| \leq \in$$
>
> An ordered field in which every Cauchy sequence converges is said to be complete.

The principal formal difference between the rationals and the reals is that, while the rationals constitute an ordered field, the reals constitute a complete ordered field. The obstacle we

face in trying to axiomatize the reals in our modified Guttag framework is that there seems at this time to be no clearly satisfactory way to formulate this notion of completeness within that framework.

In retrospect, although we may eventually find a way to formulate completeness within our framework, it may be that our present inability to do so is really a virtue, rather than a defect of our framework. The real numbers have always been really a convenient myth with respect to computer-based systems. Although we often talk in terms of real numbers, the finite character of our machines (and of ourselves) always forces us, in the end, to "round-off" our real numbers and approximate them by rationals. The problems that arise as a result of this situation are widely known (e.g. see (ZEL73)). This suggests that our present inability to formulate completeness (and thus the reals) in the framework of type algebra may, in fact, be a strength of that framework, rather than a weakness, reflecting its correctness as a model of what computer-based systems are really capable of.

APPENDIX IV BIBLIOGRAPHY

CUS76a Cushing, S., "The Formal Semantics of Quantification," U.C.L.A. Dissertation, University Microfilms, Ann Arbor, Michigan, 1976.

CUS76b _____, "The Group Structure of Quantification," U.C.L.A. Papers in Syntax, Department of Linguistics, U.C.L.A., Los Angeles, CA, 1976.

FUN74 Fundamentals of Mathematics, Vol I: Foundations of Mathematics/The Real Number System and Algebra, ed. H. Behnke, et. al., trans. S.H. Gould, The MIT Press, Cambridge, MA, 1974.

HAM76 Hamilton, M., and Zeldin, S., "AXES Syntax Description", Higher Order Software, Inc., Cambridge, MA, Dec. 1976.

LAN67 Lang, S., Algebraic Structures, Addison-Wesley Publishing Co., Reading, MA, 1967.

ZEL73 Zeldin, S., "An Example of Algorithm Comparison Using Higher Order Software Criteria," Doc. P-017, The Charles Stark Draper Laboratory, Inc., Cambridge, MA, December 6, 1973.

APPENDIX V

SAMPLE AXES SYSTEM SPECIFICATION

by W. Heath

APPENDIX V
SAMPLE AXES SYSTEM SPECIFICATION

Specification of Data Type: WORD

In what follows, the axiomatization of type WORD is given, some abstract operations are specified, and a description of the primitive and abstract operations on the type are given in Table AV-1.

DEFINE WORD;

PRIMITIVE OPERATIONS:

$word_2$ = Setspaces($word_1$,$natural_1$);

$word_2$ = Addelmt($word_1$,$natural_1$);

$word_2$ = Lastelmt($word_1$);

$word_2$ = Removeelmt($word_1$);

$natural_1$ = Nspaces($word_1$);

$natural_1$ = Nelmts($word_1$);

$boolean_1$ = Samew($word_1$,$word_2$);

AXIOMS:

WHERE 1 IS A CONSTANT NATURAL;
WHERE n,n_1,n_2 ARE NATURALS;
WHERE w,w_1,w_2 ARE WORDS;
WHERE Nullword IS A CONSTANT WORD;

Nspaces(Nullword) = Zero;

Nelmts(Nullword) = Zero;

Nspaces(Setspaces(w,n)) = n;

Nelmts(Addelmt(w,n)) = Sum(Nelmts(w),1);

Samew(w,w) = True;

Samew(Setspaces(w,n_1), Setspaces(w,n_2)) = Equal(n_1,n_2);

Samew(Addelmt(w,n_1), Addelmt(w,n_2)) = Equal(n_1,n_2);

TABLE AV-1
Description of Operations on Type WORD

OPERATION	DESCRIPTION
Nullword:	Constant value, not an operation. The value of $Word_1$ will be a word with a null string of elements and a space of length zero.
Setspaces:	$Word_1$ x Nat_1 → $Word_2$. The element string of $Word_2$ will be identical to $Word_1$. Nat_1 will be the size of the space of $Word_2$.
Addelmt:	$Word_1$ x Nat_1 → $Word_2$. Add Element. $Word_2$ will be the same as $Word_1$ except the element associated with Nat_1 will be concatinated on the end of its element string.
Lastelmt:	$Word_1$ → $Word_2$. Last Element. The element string of $Word_2$ is the last element in the string of $Word_1$. If the element string of $Word_1$ is null, the element string of $Word_2$ will be null also. $Word_2$ will have a space of size zero.
Removeelmt:	$Word_1$ → $Word_2$. Remove Element. $Word_2$ will be the same as $Word_1$ except the last element in the element string will be omitted.
Nspaces:	$Word_1$ → Nat_1. Nat_1 is the size of the space of $Word_1$.
Nelmts:	$Word_1$ → Nat_1. Nat_1 is the number of elements in the element string of $Word_1$.
Samew:	$Word_1$ x $Word_2$ → $Boolean_1$. Same Word? $Boolean_1$ has the value True if $Word_1$ and $Word_2$ are identical in element string and space size. It has the value False otherwise.
Lengthw	$Word_1$ → Nat_1. Length of Word. Nat_1 is the total length of $Word_1$, i.e., the sum of the number of elements and the size of the space of $Word_1$.
Element:	Nat_1 → $Word_1$. Creates a word with a single element corresponding to Nat_1 and a space of size zero.
Addspaces:	$Word_1$ x Nat_1 → $Word_2$. Adds Nat_1 to the size of the space of $Word_1$ to create $Word_2$. The element strings of $Word_1$ and $Word_2$ are identical.
Ndiff:	Nat_1 x Nat_2 → Nat_3. Modified subtraction defined on the natural numbers. If Nat_2 is larger than Nat_1, the value of Nat_3 is zero instead of error.

```
Lastelmt(Addelmt(w,n)) = Element(n);

Removeelmt(Addelmt(w,n)) = w;

Nelmts(Removeelmt(w)) = Ndiffz(Nelmts(w),1);

Lastelmt(Nullword) = REJECT;

Removeelmt(Nullword) = REJECT;

Samew(w1,w2) = And(Equal(Nspaces(w1)Nspaces(w2)),
     And(Samew(Lastelmt(w1), Lastelmt(w2)),
        (Samew(Removeelmt(w1), Removeelmt(w2))));
END WORD;

OPERATION:  n = Lengthw(w);
WHERE n IS A NATURAL;
WHERE w IS A WORD;
WHEREBY n = Sum(Nspaces(w),Nelmts(w));
END Lengthw;

OPERATION:  w = Element(n);
WHERE w IS A WORD;
WHERE n IS A NATURAL;
WHEREBY w = Addelmt(REJECT,n);
END Element;

OPERATION:  w2 = Addspaces(w,n);
WHERE w,w2 ARE WORDS;
WHERE n IS A NATURAL;
WHEREBY w2 = Setspaces(w,Sum(Nspaces(w),n));
END Addspaces;

OPERATION:  n3 = Ndiffz(n1,n2);
WHERE n1,n2,n3 ARE NATURALS;
EITHER m3 = Kzero(1(m1,m2)) OTHERWISE
       n3 = Ndiff(2(n1,n2));
     PARTITION OF (n1,n2) IS
     1(n1,n2)|n1<n2,
     2(n1,n2)|n1≥n2;
END Ndiffz;
```

Specification of Data Type: LINE

An algebraic specification of the line is given and some abstract operations that will be used in an example problem are defined. Table AV-2 is a description of the primitive and abstract operations that have been defined.

DEFINE LINE;

PRIMITIVE OPERATIONS:

$line_1$ = Wline($word_1$);

$word_1$ = lstword($line_1$);

$natural_1$ = Nwords($line_1$);

$boolean_1$ = Samel($line_1$,$line_2$);

$line_2$ = Head($line_1$,$natural_1$);

$line_2$ = Tail($line_1$,$natural_1$);

$line_3$ = Conc($line_1$,$line_2$);

AXIOMS:

WHERE 2,1 ARE CONSTANT NATURALS;
WHERE n,n_1,n_2 ARE NATURALS;
WHERE w,w_1,w_2 ARE WORDS;
WHERE line, $line_1$, $line_2$ ARE LINES;
WHERE Nulline IS A CONSTANT LINE;

Samel(Wline(w_1),Wline(w_2)) = Samew(w_1,w_2);

Nwords(Wline(w)) = 1;

Samel(line,line) = True;

Head(Nulline,n) = REJECT;

Tail(Nulline,n) = REJECT;

Conc(Nulline,Nulline) = REJECT;

Conc(Head(line,n),Tail(line,n)) = line;

Description of Operations on Type LINE

OPERATION	DESCRIPTION
Wline:	$Word_1 \rightarrow Line_1$. Word to Line. A type transformation. $Line_1$ is a line containing the single word $Word_1$.
1stword:	$Line_1 \rightarrow Nat_1$. First Word. $Word_1$ is the first word on $Line_1$.
Nwords:	$Line_1 \rightarrow Nat_1$. Nat_1 is the number of words on $Line_1$.
Samel:	$Line_1$ X $Line_1 \rightarrow$ Boolean . Same Line? $Boolean_1$ has the value TRUE if $Line_1$ and $Line_2$ are identical. It has the value FALSE otherwise.
Head:	$Line_1$ X $Nat_1 \rightarrow Line_2$. $Line_2$ consists of the first Nat_1-1 words on $Line_1$. If Nat_1 is less than or equal to 1, then $Line_2$ is the NullLine.
Tail:	$Line_1$ X $Nat_1 \rightarrow Line_2$. $Line_2$ is what remains of $Line_1$ after the first Nat_1-1 words are removed. If Nat_1 is greater than NWords($Line_1$), ther $Line_2$ is the NullLine.
Conc:	$Line_1$ X $Line_2 \rightarrow Line_3$. Concatination. $Line_3$ is the string of words on $Line_2$ concatinated onto the end of the string of words on $Line_1$.
Nulline	Constant value, not an operation. $Line_1$ is the line with a null string of words.
Length:	$Line_1 \rightarrow Nat_2$. Nat_2 is the sum of the lengths of each of the words within $Line_1$.
Sumw:	$Line_1$ X $Nat_1 \rightarrow Nat_2$. Sum of Word Lengths. Nat_2 is the sum of the lengths of the first Nat_1 words on $Line_1$.
Compress:	$Line_1 \rightarrow Line_2$. $Line_2$ is the same as $Line_1$ except that the size of the space preceding each word is zero for the first word and one for all others.
Compact:	$Line_1$ X $Nat_1 \rightarrow Line_2$. $Line_2$ is the same as $Line_1$ except that the first Nat_1 words are compressed.
Pad:	$Line_1$ X $Nat_2 \rightarrow Line_2$. $Line_2$ is the same as $Line_1$ except that the size of the space of each word has been increased by Nat_2.
Padeachw:	$Line_1$ X Nat_1 X $Nat_2 \rightarrow$ Line . Pad Each Word. $Line_2$ is the same as $Line_1$ except that the size of the space of the first Nat_1 words has been increased by Nat_2.

```
Nwords(Conc(line1,line2)) = Sum(Nwords(line1),Nwords(line2));

1stword(Conc(Wline(w),line)) = w;

EITHER Head(Wline(w),n) = IDENTIFY_1^2(Wline(w),1n) OTHERWISE
     Head(Wline(w),n) = K_REJECT(2n);
     PARTITION OF n IS
       1n|n=2,
       2n|n≠2;

EITHER Tail(Wline(w),n) = IDENTIFY_1^2(Wline(w),1n) OTHERWISE
     Tail(Wline(w),n) = K_REJECT(2n);
     PARTITION OF n IS
       1n|n<1,
       2n|n≥1;

Samel(Conc(line1,line2),Conc(line2,line1)) =
     OR(Samel(line1,REJECT),Samel(line2,REJECT));

Head(Conc(line1,line2),n) =
     K_REJECT(1n) AND Head(line1,2n) AND
     Conc(line1,Head(line2,Ndiffz(3n,Nwords(line1)))) AND
     IDENTIFY_1^2(Conc(line1line2),4n);
     PARTITION OF n IS
       1n|n≤1,
       2n|1<n≤Nwords(line1),
       3n|Nwords(line1)<n≤Nwords(Conc(line1,line2)),
       4n|n>Nwords(Conc(line1,line2));
```

```
Tail(Conc(line1,line2),n) =
     K_REJECT(1n) AND
     Conc(line1,Tail(line2,Ndiffz(2n,Nwords(line1)))) AND
     Conc(Tail(line1,3n),line2) AND
     IDENTIFY1^2(Conc(line1,line2)4n;
     PARTITION OF n IS
       1n|Nwords(Conc(line1,line2)< n,
       2n|Nwords(line1)<n≤Nwords(Conc(line1,line2)),
       3n|1<n≤Nwords(line1),
       4n|n≤1;

END LINE;

OPERATION:  n = Length(line1);
WHERE line1 IS A LINE;
WHERE n IS A NATURAL;
WHEREBY n = Sumw(line1,Nwords(line1));
END length;

OPERATION:  n2 = Sumw(line,n);
WHERE line1 is a LINE;
WHERE n,n2 ARE NATURALS;
EITHER n2 = K_Zero(1line1,1n) OTHERWISE
WHEREBY n2 = Sumw(2line1,2n-1) + Lengthw(Extract(2line1,2n));
     PARTITION OF (n,line1) IS
       1(n,line1)|n = Zero,
       2(n,line1)|n NOT= Zero;
END Sumw;

OPERATION:  w = Extract(line1,n);
WHERE w IS A WORD;
WHERE line1 IS A LINE;
WHERE n IS A NATURAL;
WHEREBY w = lstword(Tail(line1,n));
END Extract;
```

```
OPERATION:  line2 = Compress(line1);
WHERE line1,line2 ARE LINES;
WHEREBY line2 = Compact(line1,Nwords(line1));
END Compress;

OPERATION:  line2 = Compact(line1,n);
WHERE n IS A NATURAL;
WHERE line1,line2 ARE LINES;
WHERE 1 IS A CONSTANT LINE;
EITHER WHEREBY line2 = Wline(Setspaces(1stword(1line1),Zero))
OTHERWISE WHEREBY line2 = Conc(Compact(2line1,2n-1),
                  Wline1(Setspaces(Extract(2line1,2n),1)))
PARTITION OF (line1,n) IS
   1(line1,n)|n≤1,
   2(line1,n)|n>1;
END Compact;

OPERATION:  line2 = Pad(line1,n);
WHERE line1,line2 ARE LINES;
WHERE n IS A NATURAL;
WHEREBY line2 = Padeachw(line1,Nwords(line1),n);
END Pad;

OPERATION:  line2 = Padeachw(line1,n1,n2);
WHERE line1,line2 ARE LINES;
WHERE n1,n2 ARE NATURALS;
EITHER WHEREBY line2 = KREJECT(1(line1,1n1,1n2))
OTHERWISE WHEREBY line2 = Conc(Padeach(2line1,2n1 - 1,2n2),
                          Wline(addspaces(Extract(2line2,n1),2n2)));
PARTITION OF (line1,n1,n2) IS
  1(line1,n1,n2)|n1 = Zero,
  2(line1,n1,n2)|n1 NOT= Zero;
END Padeachw;
```

Line Justifier

The Linejustify function is designed to adjust the spacing between words of a line so that the last element of the last word of the line occurs at the specified margin. This problem, suggested by (GRI76) and modified in (HAM76), has been reformulated here to use data type LINE. The function is further constrained so that the size of the spaces between any two words on the same line will differ by no more than one, and the insertion of the larger spaces will be staggered to the left or right in alternating lines. Thus, odd (even) lines will have larger spaces separating words at the left (right) of the line. Also, the last line of a paragraph will be justified to the left only, and not to the right. Any line which cannot be compressed into the size of the margin without eliminating a minimum word spacing of size one will return an error condition. Table AV-3 lists the names and uses of variables of the Linejustify function, and Table AV-4 lists the names and uses of its subfunctions.

```
FUNCTION:  line2 = Linejustify(line1,Margin,Lpty,Ppty);
JOIN line2 = Pptychk(Compl,Margin,Lpty,Ppty) WITH
                              1     1   1
     INCLUDE Compl = Compress(line1)          ALSO

             (Margin,Lpty,Ppty) =
                 1     1    1

             CLONE1(Margin,Lpty,Ppty);
EITHER line2 = IDENTIFY1^4(1Compl,1Margin,1Lpty,1Ppty) OTHERWISE
                           1       1     1
JOIN (2Compl,2Margin,2Lpty) =
       1     1      1
     IDENTIFY1,2,3^4(2Compl,2Margin,2Lpty,2Ppty) WITH
                       1     1     1     1
     line2 = Sizechk(2Compl,2Margin,2Lpty);
                       i     1      1
```

```
PARTITION OF (Compl,Margin,Lpty,Ppty) IS
                        1     1    1

     1(Compl,Margin,Lpty,Ppty)|Ppty = True,
       1     1    1

     2(Compl,Margin,Lpty,Ppty)|Ppty = False;
       1     1    1

EITHER line2 = IDENTIFY3(1 2Compl, 1 2Margin, 1 2Lpty) OTHERWISE
                         1         1         1

EITHER line2 = Calcspaces(2 2Compl, 2 2Margin, 2 2Lpty) OTHERWISE
                          1         1         1

       line2 = K REJECT(3 2Compl, 3 2Margin, 3 2Lpty);
                         1         1         1

PARTITION OF (2Compl, 2Margin, 2Lpty) IS
               1      1       1

     1(2Compl, 2Margin, 2Lpty)|Length(2Compl) = 2Margin,
       1       1       1                1          1

     2(2Compl, 2Margin, 2Lpty)|Length(2Compl) < 2Margin,
       1       1       1                1          1

     3(2Compl, 2Margin, 2Lpty)|Length(2Compl) > 2Margin;
       1       1       1                1          1

JOIN line2 = Lptychk(Padedl,Reml, 2 2Lpty) WITH
                                  2

INCLUDE (Padedl,Reml) = F3(2 2Compl, 2 2Margin)
                           1         1

                    ALSO 2 2Lpty = Clone1(2 2Lpty);
                           2               1

WHEREBY Extraspace = Ndiffz (2 2Margin, Length(2 2Compl)),
                             1                 1

     n1 = Nwords(2 2Compl),
                 1

     Padedl = Pad(2 2Compl, Quot(Extraspace, n1)),
                  1

     Reml = Rem(Extraspace, n1);

EITHER line2 = Leftfill(1(Padedl,Reml, 2 2Lpty))
                                       2

OTHERWISE line2 = Rightfill(2(Padedl,Reml, 2 2Lpty));
                                           2
```

```
PARTITION OF (Padedl,Reml,²₂Lpty) IS
                                  2

      ¹(Padedl,Reml,²₂Lpty)|²₂Lpty = True,
                  2        2

      ²(Padedl,Reml,²₂Lpty)|²₂Lpty = False;
                  2        2

JOIN line₂ = Conc(Taill,Padleftl) WITH

JOIN(Taill,Padleftl) = F₁(¹Padedl,¹Reml)
                          1      1

WITH(¹Padedl,¹Reml) = IDENTIFY³₁,₂(¹Padedl,¹Reml,¹₂₂Lpty);
      1      1                                  2
WHEREBY R = (¹Reml,1),
                1
       Leftl = Head(¹Padedl,R),
                       1
       Taill = Tail(¹Padedl,R),
                       1
       Padleftl = Pad(Leftl,1);

JOIN line₂ = Conc(Headl,Padrightl) WITH

JOIN(Headl,Padrightl) = F₂(²Padedl,²Reml)
                          1      1

WITH(²Padedl,²Reml) = IDENTIFY³₁,₂(²Padedl,²Reml,²₂₂Lpty);
      1      1                                  2
WHEREBY Rightl = Tail(²Padedl,Ntail),
                        1
       Ntail = Nwords(²Padedl)+1-²Reml,
                      1          1
       Padrightl = Pad(Rightl,1),
       Headl = Head(²Padedl,Ntail);
                      1
WHERE line₁,line₂,Compl,Padedl,Leftl,
       Padleftl,Taill,Rightl,Padrightl,Headl ARE LINES;
WHERE Margin,Extraspace,n₁,2;
       Ntail,Reml  ARE NATURALS;
WHERE Lpty,Ppty ARE BOOLEAN;
END Linejustify;
```

TABLE AV-3
Key to Variable Names

NAME	TYPE	DESCRIPTION
$Line_2$	LINE	Output line.
$Line_1$	LINE	Input line.
Margin	NATURAL	Specified margin for output line.
Lpty	BOOLEAN	Line Parity. True for odd lines. False for even lines.
Ppty	BOOLEAN	Paragraph Parity. True for last line of paragraph. False otherwise.
Compl	LINE	Compressed Line. L_1 after it has been left justified and reduced with a space of size one between each word.
Lengthcompl	LINE	Length of CompL.
Extraspace	NATURAL	Size of space needed to expand CompL to fill Margin.
N	NATURAL	Number of words of CompL.
Quo	NATURAL	Quotient. Size of space to be divided evenly among the words of CompL.
Rem	NATURAL	Remainder. Number of spaces that must be increased by one for PadedL to fill Margin.
Padedl	LINE	Paded Line. The line CompL after Quo spaces have been inserted evenly among all its words.
Leftl	LINE	Left Line. First portion of PadedL into which Rem spaces will be inserted, one to a word.
Padleftl	LINE	Paded Left Line. The line LeftL after the size of each space has been increased by one.
Taill	LINE	Tail Line. The last portion of PadedL after LeftL has been removed from it.
Ntail	NATURAL	Number of Words to be removed from the front of PadedL so that increasing by one the size of the spaces in the remainder of the line will fill the margin.
Rightl	LINE	Right Line. Remaining portion of PadedL after the first NTail words have been removed.
Padrightl	LINE	Paded Right Line. RightL after each of its spaces has been increased by one.
Headl	LINE	Head Line. First NTail words of PadedL.

Table AV-4

Description of the Subfunction of LineJustify

NAME	DESCRIPTION
Pptychk	Paragraph Parity Check. Returns compressed line if it is the last of the paragraph.
Sizechk	Size Check. Examines the length of the compressed line to determine if there is an error or if the line already fills the margin.
Calcspaces	Calculate Spaces. Determines what space can be inserted evenly between words and calculates the remainder that must be inserted either to the left or to the right of the line.
Lptychk	Line Parity Check. Determines which side of the line to insert extra space, depending on line parity (LPty).
Leftfill	Inserts extra space to left of line.
Rightfill	Inserts extra space to right of line.

NOTE: The purpcse of this sample system problem is to show the use of AXES from the point of view of explicitly demonstrating the interfaces of the system as they would appear to an automatic analyzer. We are not attempting here to show various shorthand methods that are available to the user. Thus, this sample problem does not include, for example, the definition and the use of new abstract control structures which would both shorten the description of the system and provide for more reliable communication from the standpoint of the human analyzer. An example of such a mechanism is demonstrated by the Fail Structure (c.f. AXES Syntax Description, Section 14.0).

HIGHER ORDER SOFTWARE, INC. · 843 MASSACHUSETTS AVENUE · CAMBRIDGE, MASSACHUSETTS 02139 · (617) 661-8900

APPENDIX V BIBLIOGRAPHY

GRI76 Gries, D., "An Illustration of Current Ideas on the Deviation of Correctness Proofs and Correct Programs", in Proceedings of the 2nd International Conference on Software Engineering, San Francisco, CA, IEEE Catalog No. 76H1125-4 C, October 1976.

HAM76 Hamilton, M., and Zeldin, S., "Integrated Software Development System/Higher Order Software Conceptual Description", Version 1, Higher Order Software, Inc., Cambridge, MA, November 1976.

APPENDIX VI

SUMMARY OF THE DESCRIPTION OF AXES SYNTAX

NOMENCLATURE

In the description of AXES the following nomenclature will be used.

":=" means "is".

"{ }" means choose one of the rows contained within.

"[ ]" means the enclosed is optional.

"..." means repeat with different values as often as necessary.

In the syntax of AXES, the following nomenclature will be used.

Upper case names will designate lexical items of AXES (keywords).

"set of variables" means a list of variables possibly enclosed in parentheses.

Constants and abstract control structure names begin with an upper case character followed by zero or more lower case characters.

A variable is indicated by all lower case characters.

A value of a particular data type can be indicated by the name of the data type in lower case characters, possibly subscripted.

HIGHER ORDER SOFTWARE, INC. · 843 MASSACHUSETTS AVENUE · CAMBRIDGE, MASSACHUSETTS 02139 · (617) 661-8900

STRUCTURE

```
"STRUCTURE:"  y "=" S "(" x ");"
     declaration...
     definition...
"SYNTAX:"  user defined syntax";"
"END" S ";"
```

user defined syntax: = $connector_1$ y_1 "=" S_1 "("x_1")"...
$connector_n$ y_n "=" S_n "("x_n")"

where x, y are variables or sets of variables whose values are in the same types as the members of the ordered pairs that make up the mappings in the tuples of S;
and S is a structure name;
and $connector_i$ is a user-defined name, possibly empty;
and $y_i = S_i(x_i)$ is an unspecified mapping (see Section 10.0 for use of user-defined syntax).

The unspecified mapping names, used in definition statements within a structure, are nested subscripted names with respect to the root module name.

OPERATION

```
"OPERATION:"  y "=" L "(" x ");"
     declaration...
     definition...
"END" L ";"
```

where x, y are variables or sets of variables whose values are in the same types as the members of the ordered pairs which are the mappings,
and L is an operation name.

FUNCTION

```
"FUNCTION:"  y "=" F "(" x ");"
     declaration...
     definition...
"END" F ";"
```

where x, y are particular variables or particular sets of variables whose values are in the same types as the members of the ordered pairs which are the mappings,
and F is a function name.

DECLARATION

In $declaration_1$, x is a variable, y_1,... is a set of variables, T is a constant or variable data type name, and "S" concatenated with T denotes a plural type name.

$$declaration_1 := \text{"WHERE"} \left\{ \begin{array}{l} x \text{ "IS"} \left\{ \begin{array}{l} \left\{ \begin{array}{l} \text{"A"} \\ \text{"AN"} \end{array} \right\} [\text{"CONSTANT"}] \ T \\ \left\{ \begin{array}{l} \text{"A"} \\ \text{"AN"} \end{array} \right\} \left\{ \begin{array}{l} T_1 \ldots \\ T_1 \text{"OR"} \ldots T_n \end{array} \right\} \\ \text{"OF SOME TYPE"} \end{array} \right\} \\ y_1, \ldots \text{ "ARE"} \left\{ \begin{array}{l} [\text{"CONSTANT"}] \ T\text{"S"} \\ T_1 \ldots T\text{"S"} \\ T_1\text{"S" "OR"} \ldots T_n\text{"S} \\ \text{"OF SOME TYPES"} \\ \text{"OF THE SAME TYPE"} \end{array} \right\} \\ x \text{ "IS"} \left\{ \begin{array}{l} \text{"("}y_1\text{","}\ldots\text{")"} \\ T \\ \text{"SOME TYPE"} \end{array} \right\} \end{array} \right\} \text{";"}$$

In declaration$_2$, x is a variable or a set of variables enclosed in parentheses, ^{j}y is variable or set of variables whose values are members of the members of a partition of the set of values of the variables that x represents.

$$\text{declaration}_2 : = \text{"PARTITION OF"}\ x\ \text{"IS"} \left\{ \begin{array}{l} \text{ANY PARTITION} \\ {}^1y\ \text{"|"}\ tv_1\ \text{","} \ldots {}^iy\ \text{"|"}\ tv_i\ \text{";"} \end{array} \right\}$$

and

$$\text{true val exp}: \left\{ \begin{array}{l} F\ \text{"("}\ \text{exp}_1\ \text{")"} \\ F\ \text{"("}\ \text{exp} \ldots \text{")"} \\ \text{exp}\ F\ \text{exp} \end{array} \right\}$$

$$tv_i : = \left\{ \begin{array}{l} \text{true val exp} \\ \text{"("true val exp}_1\ \text{","} \ldots \text{true val exp}_i \text{")"} \end{array} \right\}$$

true val exp$_i$ evaluates to the boolean value True, and exp is in terms of x and values of x.

DEFINITION

In a definition, y,x are variables or sets of variables, and F is a structure, operation, or function.

$$\text{definition}_i : = \left\{ \begin{array}{l} y\ \text{"="}\ F\ \text{"("}\ x\ \text{")"} \\ \text{primitive definition} \\ \text{user-defined definition} \\ \text{mapping assertion} \end{array} \right\} \ \text{";"}$$

$$\text{Primitive definition}: = \begin{array}{l} \text{definition}_1\ \text{"AND"} \ldots \\ \text{definition}_n\ \text{";"} \end{array}$$

user-defined definition: = $\begin{matrix} \text{connector}_1\ \text{definition}_1 \ldots \\ \text{connector}_n\ \text{definition}_n \end{matrix}$

where a set of connectors is defined in a particular structure definition (see Section 8.0).

mapping assertion: = "WHEREBY" y "=" exp";"

EXPRESSIONS

exp: = $\left\{ \begin{matrix} \text{value} \\ \text{x} \\ \text{F(exp)} \\ \text{exp F exp} \\ \text{(exp)} \\ \text{exp,...} \end{matrix} \right.$

where F is an operation or a function name and
x is a variable.

CORRESPONDENCE BETWEEN INTRINSIC DATA TYPE OPERATIONS AND INFIX SYMBOLS

Operation	Symbols
Or, Por	!
And, Pand	&
Not, Pnot, Iopp, Ropp	prefix -
Same, Ident, Equal, ?Equal?, ?Iequal?, ?Requal?	=
?>?, ?I>?, ?R>?	>
Sum, Isum, Rsum	+
Idiff, Rdiff	-
Prod, Iprod, Rprod	*
Rdiv	/
Conc	!!

The precedence of AXES operators is as follows:

```
Highest
          ** prefix + -
          * /
          + -
          = < > <= >=
          &
          !
Lowest
```

Operator Associativity

If an expression contains multiple operators of equal precedence, the meaning of the expression is determined by the associativity of the operators. All prefix operators and the "**" infix operator are right-associative, and all other operators are left-associative.

Left-associative operators give priority to other operators of equal precedence to their left, while right-associative operators give priority to operators of equal precedence to their right. For example,

DATA TYPES

```
"DATA TYPE:"  name ";"
"PRIMITIVE OPERATIONS;"
    primitive operations...
"AXIOMS;"
    declaration...
    assertion (about a type)...
"END" name";"
```

where
(1) name is the abstract data type name.

primitive operation:= typename$_i$ "=" P_i "(" typename$_j$,..."");"

where typename$_k$ is a data type name in lower-case characters and k is an integer, possibly empty, and P_i is a primitive operation name.

assertion (about a type): = $\left\{\begin{array}{l}\text{definition}_1 \\ F"("exp_1")"\end{array}\right\}$ "=" $\left\{\begin{array}{c}\text{definition}_2 \\ exp_2\end{array}\right\}$ ";"

intrinsic types: = $\left\{\begin{array}{l}\text{boolean} \\ \text{natural} \\ \text{integer} \\ \text{rational} \\ \text{property (of T)} \\ \text{set (of T)} \\ \text{line}\end{array}\right\}$ value: = $\left\{\begin{array}{l}\text{boolean value} \\ \text{natural value} \\ \text{integer value} \\ \text{rational value} \\ \text{property (of T) value} \\ \text{set (of T) value} \\ \text{line value} \\ \text{extrinsic value}\end{array}\right\}$

boolean value: = $\left\{\begin{array}{l}\text{True} \\ \text{False}\end{array}\right\}$

natural value: = $\left\{\begin{array}{l}0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9\end{array}\right\}$ ...

integer value: = $\{^{+}_{-}\}$ natural

rational value: = $\begin{Bmatrix} integer_1. \\ integer_1.integer_2 \end{Bmatrix}$ ["E" integer]

property (of T) value: : = "PROPERTY OF" t "IN" T"|"true val exp_1

set (of T) value: = $\begin{cases} \{value_1,...\} \\ \text{"SET OF" t "IN" T "|" true val } exp_1 \end{cases}$

line value: = 'any finite string of symbols possibly empty'

Extrinsic data type values are defined as "'CONSTANT' T" using a $declaration_1$ statement (Section 9.0).

"DERIVED OPERATION:" y "=" D "(" x ");"

declaration...

assertion(about D)...

"END" D ";"

where x,y are variables or sets of variables and D is a derived operation name.

assertion(about D): = $\begin{Bmatrix} definition_1 \\ F_1"("exp_1")" \end{Bmatrix}$ "=" $\begin{Bmatrix} definition_2 \\ F_2"("exp_2")" \end{Bmatrix}$ ";"